PROFILE

Name Fahad Ali

Subject Java / DSA. (Lecture 1 – 65)

Academic Year 2022-23

Address (Hyd) - Ind.

Branch (Start → Java).

Professor in Charge Durga Sir.

Institute / University / School Durgasoft classes. By :-

Semester (Start - 21 May 22 – 25 Aug 2022 (1:25 PM))

Contact Hyderabad.

Inspirational & Motivational Thoughts

Things work out best for those who make the best of how things work out.
~ John Wooden

If you are not willing to risk the usual you will have to settle for the ordinary.
~ Jim Rohn

All our dreams can come true if we have the courage to pursue them.
~ Walt Disney

Success is walking from failure to failure with no loss of enthusiasm.
~ Winston Churchill

Just when the caterpillar thought the world was ending, he turned into a butterfly.
~ Proverb

Try not to become a person of success, but rather try to become a person of value.
~ Albert Einstein

Date 21 May 2022.
Saturday

Java

Language Fundamentals:-

1) Identifiers
2) Reserved words.
3) Data Types
4) literals
5) Arrays
6) Types of Variables
7) Var-arg methods.
8) main method.
9) Command line arguments.
10) Java coding standards.

(1) Identifiers:- It is used for identification purpose.

OR

name in Java program is by default considered as identifier, which can be used for identification purpose.

ex:- It can be a class name, method name, variable name, and label name → l1:, l2:.

```
class Test → 1
{
    public static void main (String[] args)
    {
        int x = 10;
    }
}
```

2 → main
3 → predefined Java class name
4 → user defined name
5 → x
6 → String

⇒ 5 identifiers.

Rule to naming Java identifiers :-

1) The only allowed characters are
   - a to z
   - A to Z
   - 0 to 9
   - $, _ (underscore)

   If we are using any other character we will get compile time error.

ex: total_number, total#. 123total
✓ valid ↵ Invalid. Invalid.

2) Identifiers can't start with digits.
   ex: total123 , 123total
   valid Invalid.

3) Java identifiers are case-sensitive.
   Of course, Java lang. itself treated as ~~yes~~ case-sensitive programming language.

   ex:-
   ```
   class Test
   {
       int number = 10;
       int Number = 10;
       int NUMBER = 10;
   }
   ```
   we can differentiate w.r.t case

4) There is no length limit for Java identifiers, but it is not recommended to take

too lengthy identifies.

5) We can't use reserved words as identifiers.

Ex:
```
int x = 10;   → valid
int if = 20;  → Invalid.
```
if → Reserved words.

6) class Test ⇐ Ex
```
{
    public static void main(String[] args)
    {
        int String = 000;
        System.out.printen(String);
    }
}
```
String → class name

```
int runnable = 999;
System.out.printen(runnable);
```
runnable → interface name

* All predefine Java class names and Interface names we can use as identifiers. but not recommended.

Even though it is valid but it is not a good programming practice, because it reduces readability and creates confusion.

★ 6 - Rules To Naming identifiers :-

1) Alphabet, Symbols, numbers, $, _
2) Can't starts with digit. 0-9.
3) Case - Sensitive.
4) No - length limits for identifiers. (Java)
5) Can't used reserved words as identifiers.
6) All predefine Java class & interface name happy we can use as identifiers.

Date / /

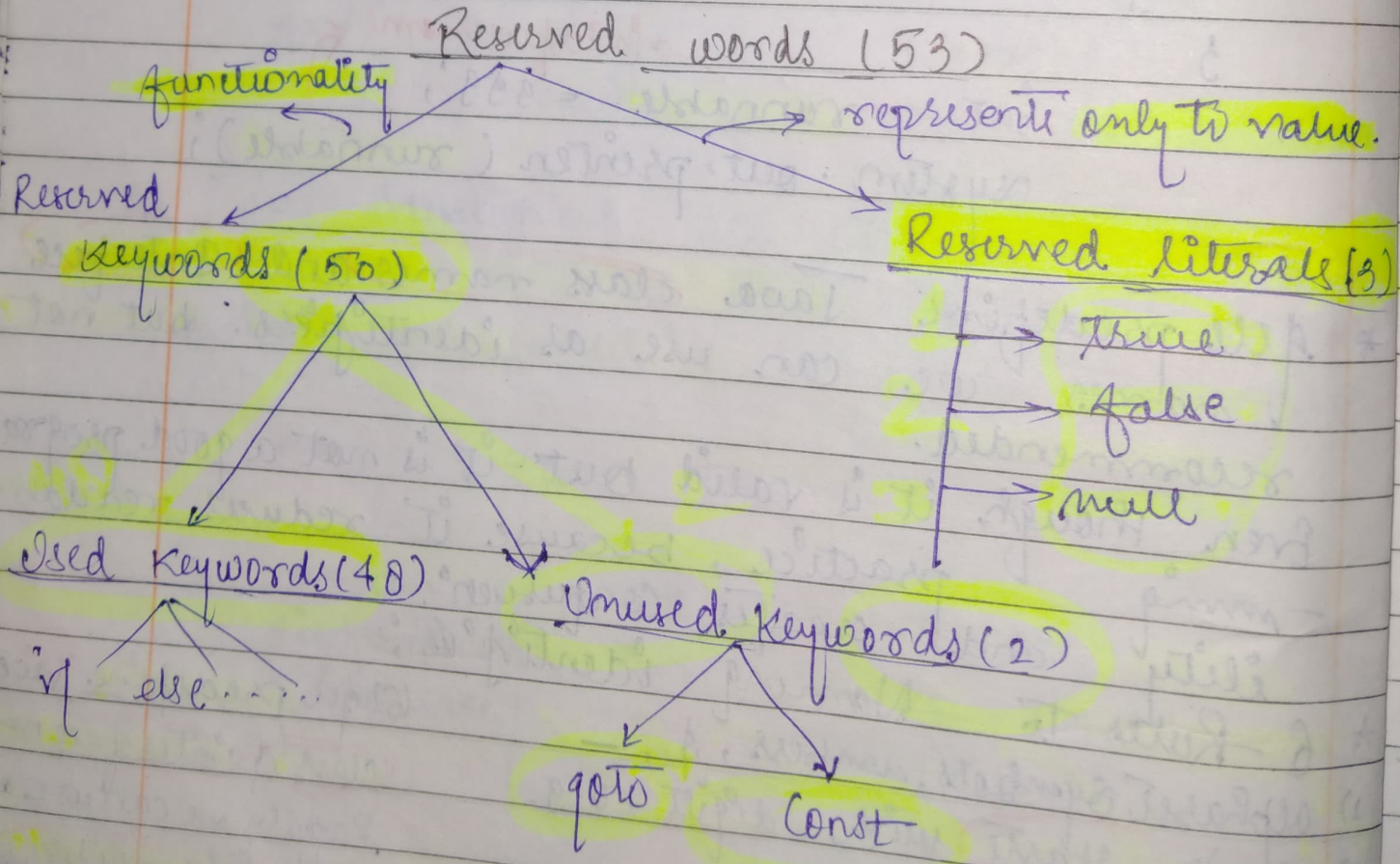
④

Which of following is valid identifiers:

1) total_number ✓
2) total # ✗
3) 123 total ✗
4) total 123 ✓
5) cash ✗
6) _$_$_$_$_ ✓
7) all @ hands. ✗
8) Java 2 share. ✓

1) Integer ✓
2) Int ✓
3) int ✗

(5)(a) Reserved words: In Java some words are reserved words to represent some meaning or functionality such types of words are called reserved words.



Note: If reserved words related to functionality called reserved keywords.
: If reserved words related to or only to represent value called reserved literals.

* Reserved Keywords for Data Types :- (8)

1) byte
2) short
3) int
4) long
5) float
6) double
7) boolean
8) char.

Integer
Decimal
T/F
character

* Reserved Keywords for flowcontrol :- (11)

1) if
2) else — conditional Stmt
3) switch : 4) case, 5) default
6) do
7) while → looping
8) for
9) break
10) continue
11) return.

Keywords for modifiers (11)

1) public
2) private
3) protected
4) static
5) final
6) abstract
7) synchronized
8) native
9) strictfp (1.2v)
10) transient
11) volatile.

Date / /

(6) Keywords for exception handling :- (6)

1) try
2) catch
3) finally
4) throws
5) assert (1.4v) → used for debugging purpose.
6) throw.

(6) * class related Keywords :-

1) class
2) interface
3) extends
4) implements
5) package. } → package / library
6) import.

* Object related Keywords :- (4)

1) new ✓
2) instanceof ✓
3) super ✓
4) this. ✓

Note :- Delete :- Keywords are not there in Java, because destruction of useless objects is a responsibility of garbage collector. (3)

CLASSTIME Pg. No.
Date / /

Returntype Keyword:-

1) Void

Note:- In Java returntype is mandatory.
If a method won't return anything, then we have to declare that method with Void returntype.

But in C, language returntype is Optional and the default returntype is int.

Unused Keywords:- (2)

1) goto
2) Const → Final in Java

1) goto:- Uses of goto created several problems in old language and hence some people ban. this keywords in Java.

2) Const:- Use final instead of Const.

Note:- goto and Const are unused keywords and if we are trying to use we will get Compile time error.

(3) Reserved literals:-

true } Values for boolean datatypes.
false }

null → default value for object refrenc

* enum (1.5 v)

"We can use enum to define a group of named constant."

```
enum month
{
    JAN, FEB, .... DEC;
}
```

```
enum Beer
{
    KF, KO, RC, ... FO;
}
```

Conclusions:

(1) All 53 reserved words contain only alphabetical character / alphabet symbols.

(2) All 53 reserved words contain only lowercase alphabet symbols. in Java.

- In Java we have only new keywords, and there is no delete keyword, Because destruction of useless objects is the responsibility of garbage collector.

(3) The following are new keywords in Java.

1) strictfp → 1.2 v
2) assert → 1.4 v
3) enum → 1.5 v

(4) 1) strictfp but not strictFp
2) instanceof but not instanceOf
3) synchronized but not synchronize

4) extends but not extend
5) implements but not implement
6) import but not imports.
7) const but not constant.

Which of the following list contains only Java reserved words:

1) new, delete
2) goto, constant
3) break, continue, return, exit.
4) final, finally, finalize.
5) throw, throws, thrown.
6) notify, notifyAll
7) implements, extends, imports.
8) sizeOf, instanceof
9) instanceOf, strictFp.
10) byte, short, Int
11) None of the above

Which of the following are Java reserved words:

public ⤻
static ⤻ } reserved keywords
void ⤻

String → name of predefined class.
args. → name of variable
main → its method name. not reserved word.

CLASSTIME Pg. No.
Date 25/05/22 Monday

Lecture 2

(3) Data Types :-

* In Java every variable & every expression has some type.
* Each and every datatype is clearly defined
* Every assignment should be checked by compiler for type compatibility.
* Because of above reasons we can conclude Java lang. is strongly typed progs. lang.

View about OOPs :-

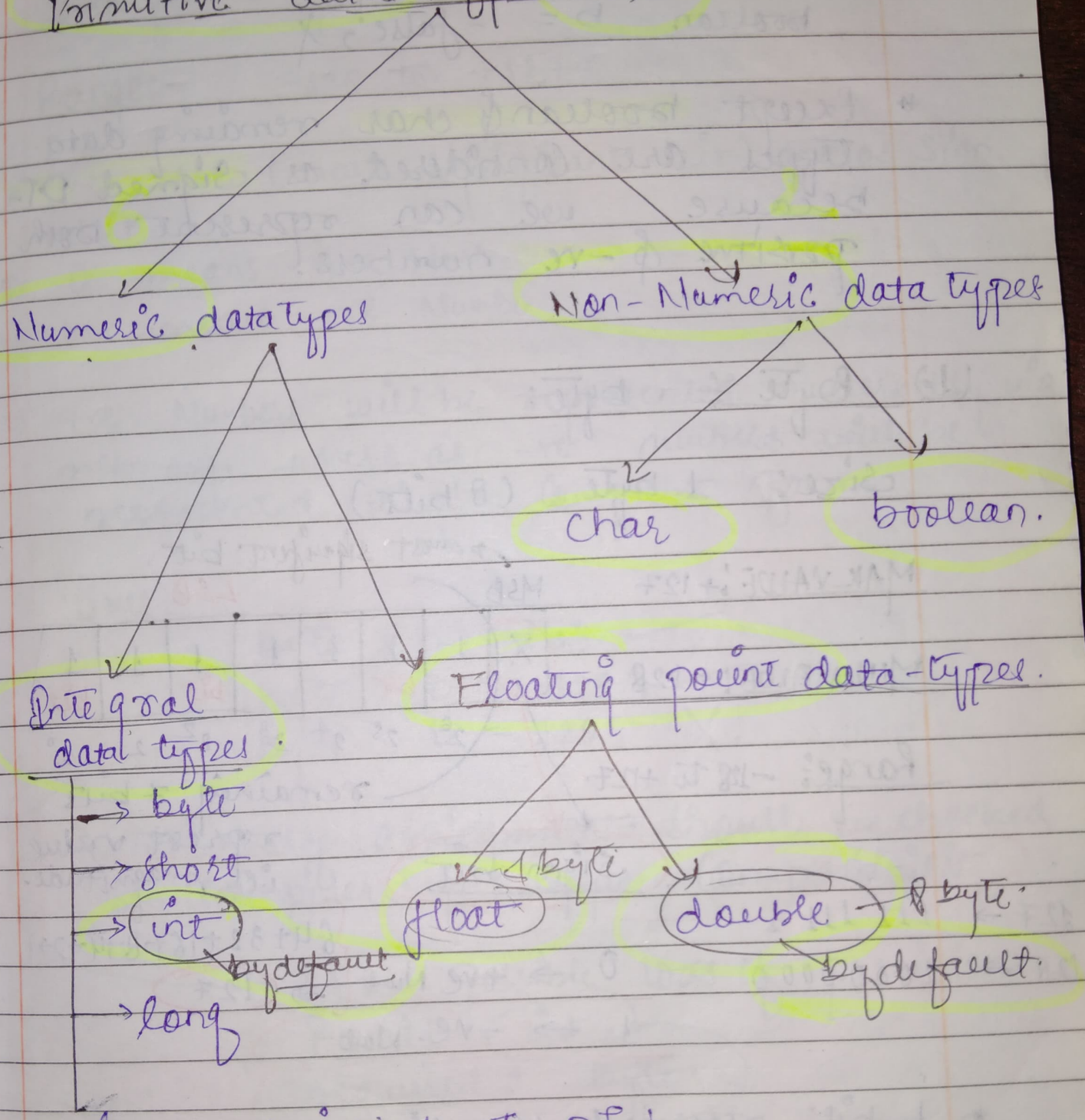
Java is not considered as pure OOPs lang. because several OOPs features are not satisfied by Java. like - Operator overloading, multiple inheritance etc.

Moreover we are depending on primitive datatypes which are non objects.

Conclusions :-

1) Java is strongly typed programg. lang.
2) Java is not pure OOPs prog. lang.

Primitive data types (8) :-



Common point about DT :-

int x = +10; ✓
int x = -10; ✓
double d = -10.5; ✓

for numbers its valid (+ve, -ve) but for char & boolean its not valid.

```
char    ch = -'a';  X
boolean b = -false; X
```

* Except boolean & char remaing data types are considered as Signed D.T. because we can represent both positive & -ve numbers!

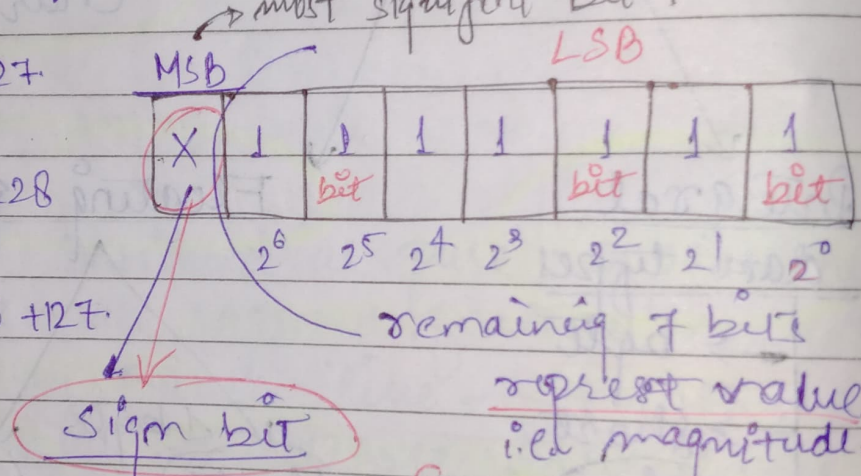
(1) Byte :- byte:

Size: 1 byte (8 bits)

MAX VALUE: +127

MIN VALUE: -128

Range: -128 to +127.



127 → 111 111 1

128 → 10000000

0 → +ve Numb

1 → -ve. Numb

64+32+16+8+4+2+1 → 127.

* 1-bit reserved for sign.

Note :- +ve Numbers will be represented directly in the memory. But, -ve Numbrs will be represented in the 2's Complement form. By using 2's complement 1 bit are

enough to represent -128. (will see in detail).

Range:- -128 to +127.

* The MSB (most significant bit) act as Sign bit.

1) 0 means +ve Numb.
2) 1 means -ve Numbe

3) +ve Numbers will be represented directly in a memory, where as -ve numbers will be represented in 2's Complement form.

Ques:-

```
byte b = 10;   ✓
byte b = 127;  ✓
byte b = 128;  X.
```

Each & every assignment should be checked by compiler for type Compatibility.

→ CE:- possible loss of precision.
Found:- int.
required: byte.

→ because every integral number by default is int type.

Type mismatch: Can not convert from int to byte.

(14)

* byte b = 10.5; ✗

CE: possible L.P
found: double
req:- byte.

* byte b = true; ✗
No. bool

CE: incompatible types
found: boolean.
req: byte.

* byte b = durga; ✗

value

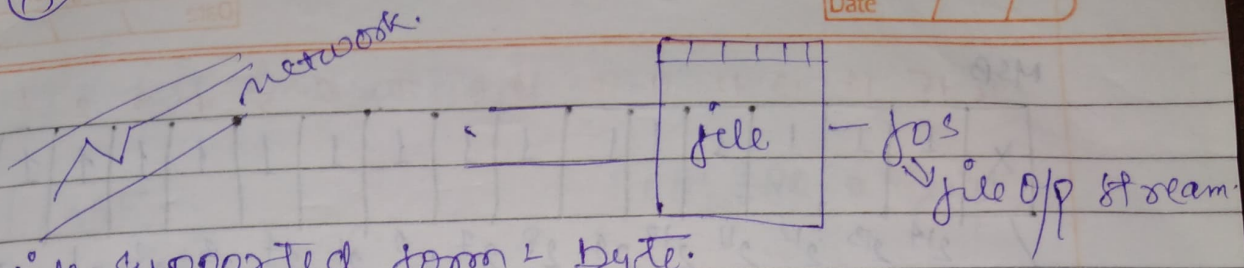
CE: incompatible types.
found: java.lang.string
req: byte.

Note:- Version To version small changes in ~~Runtime error~~ and compile time error & runtime exception.

Where we use byte? (Best choice):
If we want to handle data in the form of streams.

If want to write data to file or snd data across network.

If we want to handle data in terms of ~~byte~~ streams, then compulsory we should go for byte data types.



file supported form :- byte.
Network supported form :- byte.

Two types of streams :-

1) character streams.
2) byte streams.

* fos (class) can be used to write binary data to file.

fos.write(byte[] b);
(method) → byte array.

* byte is a best choice if we want to handle data in terms of streams. either from the file or from the network. (file supported form or network supported form is byte).

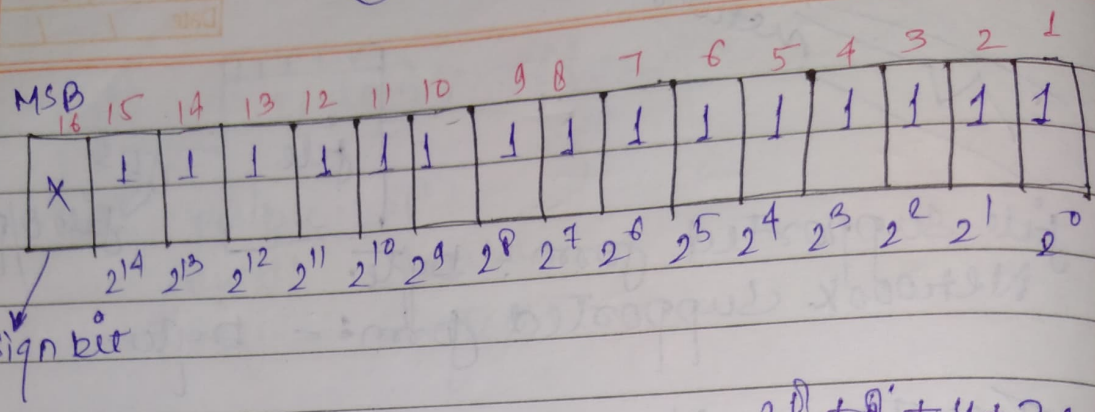
(2) short : The most rarely used D.T in Java.

size : 2 bytes (16 bits).

Range : -2^{15} to $2^{15}-1$ (Min value, Max value, -1 → sign bit)

[-32,768 to 32767]

value of 16 bit
1 bit reserved for sign.



$\Rightarrow 16384 + 8192 + \ldots\ldots 2^0 + 8 + 4 + 2 + 1$

$\Rightarrow 32768 - 1$

$\Rightarrow$ 32767

(1) short s = 32767; ✓

short s = 32768; ✗ CE: PLP
found: int
req: short.

(2) short s = 10.5; ✗
CE: PLP
found: double.
req: short

(3) short s = true; ✗
CE: in-compatible types.
found: boolean.
req: short.

Why short is not more in use?

Java come in 1995 at that time DOS up are in use.

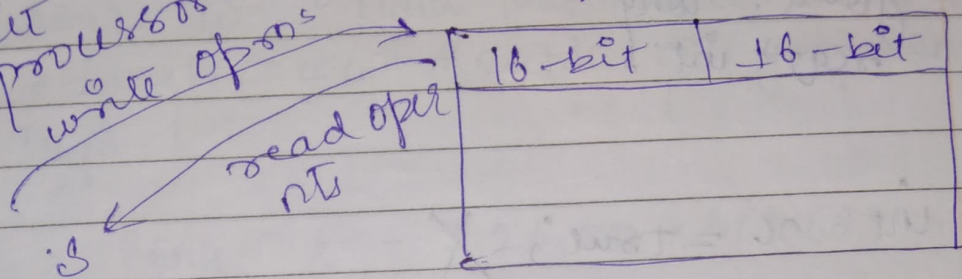
16-bit processor very popular at that time 16-bit processor means

16 bit instructions.

1995.

8085 up 8086 up

16-bit processors.



short is 16-bit

short D.T is best suitable for 16 bit processors like 8086. But these processors are completely outdated. and hence corresponding short D.T is also outd-ated D.T.

(3) Int (int):-

Size: 4 bytes (32 bits)

Range: -2^{31} to $2^{31}-1$

[-21,4748,3648 to 2147483647]

The most commonly used D.T in java is int.

1) int x = 2147483647; ✓

2) int x = 2147483648; ✗ integer no too large.

CE!:-

Note: Every integral NO. by default going to be considered as int type. So error in int will be different

Special case

int x = 2147483648l; → long. → treated as long

CE: :
PLP
found: long
req: int

int x = true; ✗

CE: incompatible types
found: boolean
req: int.

Lecture 3 :

Some time int range may not enough to hold big values.

like - I want to spe. represent the amount of distance travelled by light in 1000 days.

Speed of light ⇒ 3×10^8 m/s.

1,26,000 miles/sec.

long l ⇒ 1,26,000 × 60 × 60 × 24 × 1000 miles.
⇒ long.

long: Some time int may not enough to hold big values, then we should go

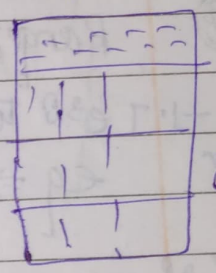
for long type.

ex1: The amount of distance travelled by light in 1000 days. to hold this value int may not enough. we should go for long data type.

```
eg: long l = 1,26,000 * 60 * 60 * 24 * 1000;
```

ex2: The no of characters present in a big file may exceed int range. Hence the return type of length() method is long but not int.

```
long l = f.length();
```



Size: 8 bytes (64 bits)

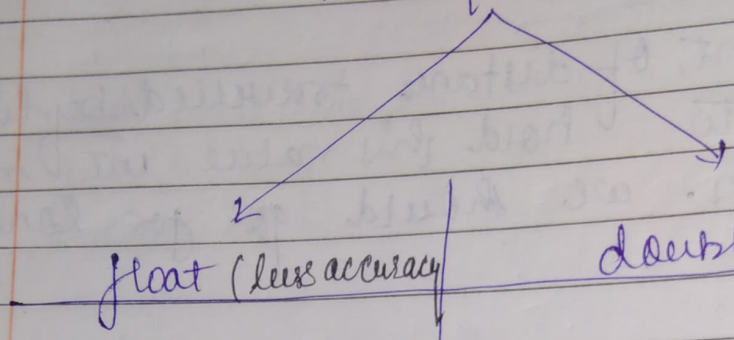
Range: -2^{63} to $2^{63}-1$

Note: All the above data-types (byte, short, int, long) meant for representing integral values. If we want to represent floating point values then we should go for floating point data types.

* Floating point Data types: (2)

1) ~~char~~ float
2) double.

Floating Point data types



float (less accuracy)	double. (more).
① 5 to 6	② 14 to 15 digit accuracy.
② Single precision	③ double precision
③ Size: 4 bytes	④ Size: 8 bytes.
④ Range:	④ Range:
$-1.7e^{38}$ to $1.7e^{38}$	$-3.4e^{308}$ to $3.4e^{308}$

eg ⇒ $\frac{10.3}{3}$ ⇒ 3.3333333.....

1.7×10^{38}

⑤ ★ If we want 5 to 6 decimal places of accuracy then we should go for float.

⑤ If we want 14 to 15 decimal places of accuracy. then we should go for double

⑥ float follows single precision

Correction :-

Range: float :-

$-3.4e^{38}$ to $3.4e^{38}$

double :- $-1.7e^{308}$ to $1.7e^{308}$

* char (DT) :- Old Languages (like C, C++) are ASCII code base and the no. of different allowed ASCII code characters are ≤ 256.

To represent these 256 characters 8 bits are enough. Hence the size of char in old languages is 1 byte.

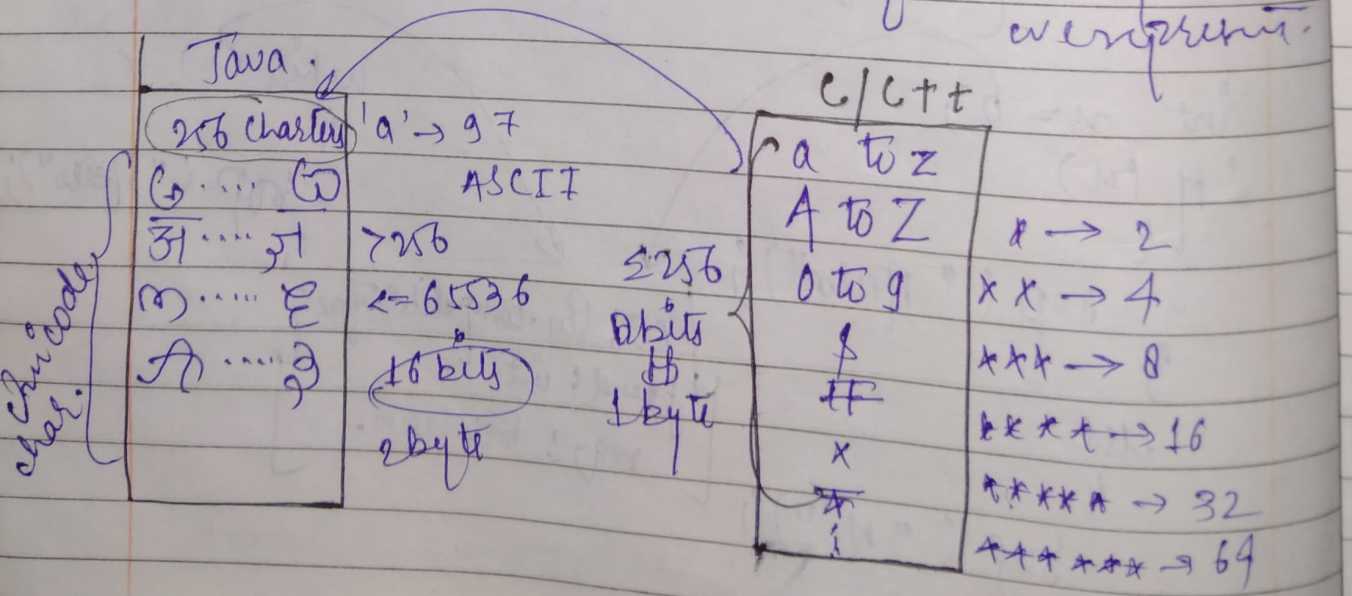
But Java is Unicode base and the no. of different Unicode characters are > 256 and ≤ 65535.

To represent these many characters 8 bits may not enough. Compulsory we should go for 16 bit. Hence the size of char in Java is 2 bytes.

Size :- 2 bytes

Range :- 0 - to [65,535]

↳ Only this no. of char are represent.



Summary of Java primitive data types :-

Data type	size	Range	wrapper class	default value.
byte	1 byte	-2^7 to 2^7-1 (-128 to 127)	Byte	0
short	2 bytes	-2^{15} to $2^{15}-1$ (-32768 to 32767)	Short	0
int	4 bytes	-2^{31} to $2^{31}-1$ (-2147483648 to 2147483647)	Integer	0
long	8 bytes	-2^{63} to $2^{63}-1$	Long	0
float	4 bytes	$-3.4e38$ to $3.4 \times e^{38}$	Float	0.0
double	8 bytes	-1.7308 to 1.7308	Double	0.0
~~boolean~~ ~~character~~	N.A	N.A [But allowed values are true/false]	Boolean	false.
char.	2 bytes	0 to 65535	Character	0 [represents space character]

Integral

NOTE :- Null is the default value for object reference and we can't apply for primitives D.T. if we are trying to use for primitive then we will get compile time error.

```
char ch = null;
```

CE! Incompatible types
found : <null type>
req : char

CLASSTIME Pg. No.
Date 25 May 22 Wednesday.

Lecture 4 :-

Literals :-

```
int x = 10;
```

datatype/keyword → int
name of variable/identifier → x
constant value/literal → 10

A constant value which can be assigned to the variable is called literal.
ex:- int x = 10;
↳ literal.

Integral literal

In how many ways we can specify literal value for the integral data types.

① decimal literals; (base - 10)
int x = 10; 0 → 9
↳ decimal form.

② Octal form = (base 8)
(0 — 7) — allowed digit
int x = 010;
The no is prefixed with zero called Octal

③ Hexa-decimal form = (base - 16)
int x = 0X10; → 0 to 9
→ a to f
a ↓ 10
f ↓ 15

CLASSTIME Pg. No.
Date / /

(1) **Integral literals** :- for integral D.T (byte, short, int, long) we can specify literal value in the following ways.

(1) decimal literals (base - 10)
allowed digits are 0 to 9.
ex:- int x = 10;

(2) Octal literals: (base - 8)
allowed digits are - 0 to 7
ex:- int x = 010; - literal value should be prefixed with zero.

(3) Hexa - decimal form literals :- (base - 16).
allowed digits are - 0 to 9, a to f

For extra digits (a to f) we can use both lower case and uppercase characters. this is one of very few areas where Java is not case-sensitive.

The literal value should be prefixed with
0x or 0X are allowed.

ex:- int x = 0X10;

These are only possible ways to specify literal value for integral data-types

Q: Which of the following declarations are valid.

int x = 10; ✓
int x = 0786; ✗ (∵ integer no too large.
int x = 0777; ✓

int x = 0XFace; ✓
int x = 0XBeef; ✓
int x = 0XBeer; ✗

int x = 0786; ✗ CE: integer no too large

Q: ex-2
class Test
{
  p s v main(String[] args)
  {
    int x = 10;
    int y = 010;
    int z = 0X10;
    sopln(x+"..."+y+"..."+z);
  }
}

3.

O/p: $(10)_8 = (?)_{10}$
$0\times8^0 + 1\times8^1 \Rightarrow 8$

$(10)_{16} = (?)_{10}$

$0\times16^0 + 1\times16^1 = 16$

O/p: (10...8....16) – Because JVM always provide value in decimal form only always by default.

* By default every integral literal (decimal, octal, hexadecimal) is of int type but we can specify

explicitly as long type by suffixed with 'l' or 'L'.

10
010
0X10
→ by default int type

10l
010L
0X10L
→ long type.

int x = 10; ✓
long l = 10L; ✓ long 8byte.

int x = 10L; ✗ → CE: PLP found: long req: int. → This error occur because we can not assigned 8byte value to 4byte variable.
long l = 10; ✓

4byte — 8byte — 4byte.

There is no direct way to specify byte and short literals explicitly. But implicitly we can specify. whenever we are assigning integral literal to the byte variable until the value with in the range of byte then compiler treats it automatically as byte literal.

Similarly short literal also.

Ex: byte b = 10;
byte b = 127;
byte b = 128; CE: PLP found: int req: byte

10b
10s

ex: short s = 32767; ✓

short s = 32768; ✗ value not in range of short

CE: PLP
found: int
req: short

(22) floating point literals:

float f = 123.456; ✗ CE: PLP
4byte double-8byte found: double
req: float.

How we can specify explicitly as float type

✓ float f = 123.456F;

✓ double d = 123.456;

By default every floating point literal is of double type and hence we can't assigned directly to the float variable.

But we can specify floating point literal as float type by suffixing with 'f' or 'F'.

float f = 123.456F; ✓
double d = 123.456; ✓

We can specify explicitly floating point literal as double type by suffixed with 'd' or 'D'. Of course this convention is not required.

ex → double d = 123.456D; ✓

float f = 123.456d; ✗
CE: PLP
found: double
req: float

✓ double d = 123.456; → valid
✓ double d = 0123.456; → it is decimal literal but not octal literal → return → decimal.
✗ double d = 0X123.456; → Hexa. → return → hexadecimal → CE: malformed floating point literal.

Note: floating point literals we can only specify only in decimal form. → 26 may — Thu - start

Note: We can specify floating point literals only in decimal form, and we can't specify in octal & hexadecimal forms.

* We can assign integral literal directly to floating point variables and that integral literal can be specified either in decimal, octal, or hexa decimal forms.

✗ double d = 0786;
CE: integer number too large

✓ double d = 0XFaceOff;

✓ double d = 0706.0;

✗ double d = 0XFace.0;

✓ double d = 10;

✓ double d = 0777;

double d = 10;
sop(d) = 10.0;

* we cant assign floating point literals to integral types.

✓ double d = 10;
int x = 10.0; ✗

CE: PLP
found: double
req: int

we can specify floating point literal even in exponential form ("scientific notation").

✓ double d = 1.2e3;
Sops(d) = 1200.0

✗ float f = 1.2e3;

CE: PLP
found: double
req: float.

✓ float f = 1.2e3F;

1.2e3
⇒ 1.2 × 10^3
⇒ 1.2 × 1000
⇒ 1200.0 ✓

(3)* Boolean literals :- true / false

* The only allowed values for boolean datatypes are true / false.

✓ boolean b = true;

✗ boolean b = 0; → CE: In-Compatible types found: int req: boolean.

✗ boolean b = True;

✗ boolean b = "true";

→ CE: cannot find symbol. found: variable True location req: class Test

→ CE: In-Compatible types: found: java.lang.String req: boolean.

```
int x = 0;
if (x)          --> CE! In Compatible types
{                   found! int
   sops ("Hi");     req: boolean.
}
else
{
   sops ("Hello");
}
```

OR

```
while (1)       --> CE! In
{
   sops ("Hello");
}
```

(5) ★ Lecture - 5 | Literals Part - 2 (26-May-2022) Thursday

(4) character : we can specifying char literal as single charater with in single Quotes.

ex: char ch = 'a';

```
char ch = a;    --> CE: Can not find symbol
                    symbol: variable a
                    location: class Test

char ch = "a";
      --> CE:- in compatible types
          found: java.lang.String
          req: char

char ch = 'ab';
      --> CE1: Unclosed char literal
          CE2:     "          "
```

JAVA BY D.S D.S

CE:- not a (java) statement.

(2) We can specify char literal as integral literal which represents Unicode value of the character and that integral literal literal can be specified either in octal, decimal, hexadecimal forms. but allowed range is 0 to 65535.

ex:- char ch = 97;
sops(ch); = a

char ch = 0XFace; ✓
char ch = 0777; ✓
char ch = 65535; ✓
✗ char ch = 65536; → CE: PLP found: int
req = char.

(3) We can represent char literal in Unicode representation which is nothing but single

'\uXXXX'

4-digit hexa-decimal number.

16	97
16	6 — 1
	(61)

eg:- char ch = '\u0061';
sops(ch); ⇒ a

Interview concept:

char ch = 97;
sop(ch); ⇒ a
char ch = 197;
O/P ⇒ +
char ch = 1970; O/P ⇒ ?

char ch = 1971; ⇒ o/p = ?
char ch ⇒ 1971O; ⇒ o/p = ?

Why? always ?

Ans: opt 1 The corresponding character not available or not defined so there is not character associated with 1970, 1970.

Ans: opt 2: The corresponding character are available & ass present but there is no font available in this system. So system can't print ?.
(www.unicode.org).

(4) Every escape character is a valid char literal.

char ch = '\n'; ✓
char ch = '\t'; ✓
char ch = '\m'; ✓
↳ CE: illegal escape character.

In Java

	Escape character	Description
1.	\n	→ New line
	\t	→ Horizontal Tab
	\r	→ Carriage return.
	\b	→ back space
	\f	→ form feed
	\'	→ single quote
	\"	→ double quote

\\ → Back Slash Symbol.

sops ("This is \' Symbol");

sops ("This is \" Symbol");

sops ("This is \\ character");

ex: filepath :(c:\\durga.class..)

Q which of the following are valid : 6:2pm

char ch = 65536; X
char ch = 0XBeer; X
char ch = '\uface'; X
char ch = '\ubeef'; X
char ch = '\m'; X
char ch = '\iface'; X

(5) String Literal :-

String s = "durga";

Any sequence of characters with in double quotes is treated as String literals.

* 1.7 ver enhancement wrt literals.

(1) Binary literals.
(2) Use of _ → Underscore.
for integral D.T → byte, short, int, long,
until 1.6 version we can specify literal value in the following ways.
1) decimal 2) Octal 3) Hexa-decimal form.

But from 1.7v onwards we can specify literal value even in binary form also.

allowed digits are 0 and 1.
literal value should be prefixed with 0b, 0B.

ex: int x = 0B1111;
sop(x) = 15.

12) Usage of Underscore (_) Symbols in numeric literals.

from 1.7v onwards we can use _ symbol between digits of numeric literals.

double d = 123456.789;

double d = 1_23_456.7_8_9;
double d = 123_456.7_8_9;

it increases code readability. main advntge of this approach.

At the time of compilation these (_) symbols will be removed automatically hence after compilation the above lines will become.

double d = 123456.789;

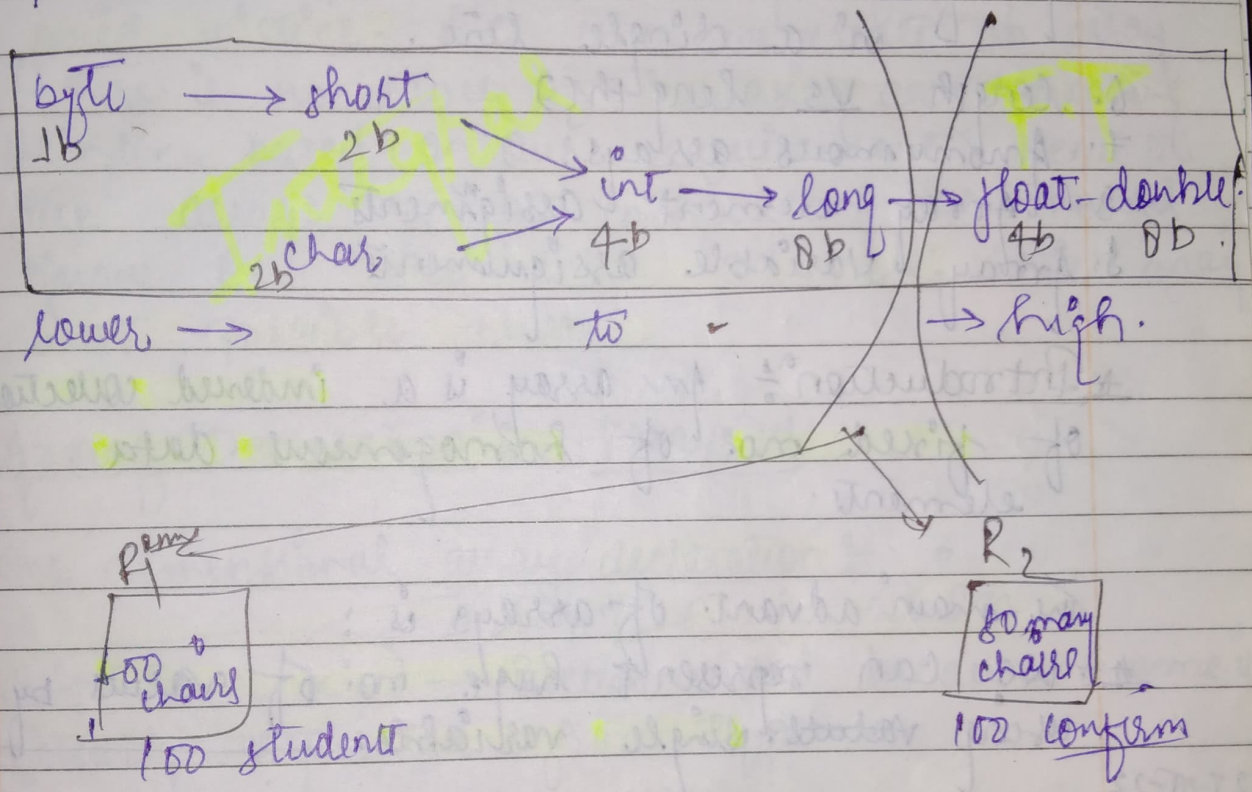
We can use more than 1 underscore symbol also b/w the digits.

Conclusion

double d = 1__23_4_3__6_7_8_9;
double d = 1___2__3_4_5_6_7_8_9;

✗
```
double d = _1_2_3_456.7_0_9;
double d = 1_2_3_456_.7_0_9;
double d = 1_23_456.7_0_9_;
```
→ Misuse

* We can use (_) symbol only in the digits. if we are using anywhere else we will get compilation errors.



Note :- 8-byte long value we can assigning to 4 byte float var. because both are following different memory representation internally.

```
float f = 10l;
sop(f); → 10.0. ✓
```

21 May – 26 May 2022 (end)
5 days

Lecture 6 / Arrays

1. Introduction
2. Array Declaration
3. Array Creation
4. Array Initialization
5. Array Declaration, creation & Initialization in a single line.
6. length Vs length()
7. Anonymous arrays.
8. Array element assignments
9. Array variable assignments

* Introduction :- An array is a indexed collection of fixed no. of homogeneous data elements.

The main advant. of arrays is :

* We can represent multiple no. of values by using single variable.

27-05-22
10:29 am

```
int x = 10;
int y = 20;
int y = 30;
:
:
:
:
```

```
int [] x = new int [10000];
```

x → [10 | 20 | 30 | . . . . .]
0 1 2

CLASSTIME Pg. No.
Date / /

Limitations of array :-

1. Array are fixed in size.
2. Array can hold only one type (homogenous) data elements.

→ So that readability of the code will be improved But the main disadvantage of arrays is fixed in size. i.e. Once we create an array there is no chance of increasing and decreasing the size based on our requirement. Hence to use arrays concept compulsory we should know the size in advance, which may not possible always.

* Array Declaration :- (3 ways) :-

i) One dimensional array declaration :-

Array Declaration

1) `int[] x;` → Recommended because name is clearly seperated from type.
2) `int [] x;` ✓
3) `int x[];`

1.D.T array type | array symbol / dimension | name of the array.

Conclusion :-

1) At the time of declaration we can't specify the size, otherwise we will get compile time error.

`int[6] x;` ✗ `int[] x;` ✓

(2) Two dimensional array declaration : (6 ways)

Valid

1) `int [][] x;`
2) `[][] x;`
3) `int x[][];`
4) `int [] [] x;`
5) `int [] x[];`
6) `int [][] x;`

* **Conclusion/concept regarding dimension wise :**

a, b → represents dimension

- ✓ `int [] a, b;` → a = 1, b = 1
- ✓ `int [] a[], b;` → a → 2, b → 1
- ✓ `int [] a[], b[];` → a = 2, b = 2
- ✓ `int [] [] a, b;` → a → 2, b → 2

⇓ This space is ignored by compiler!

Special

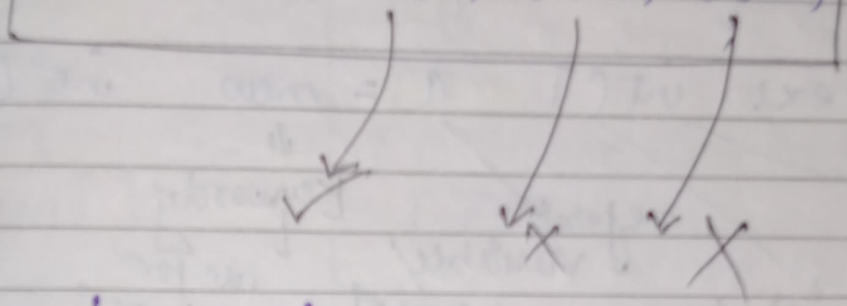
- ✓ `int [] [] a, b[];` → a → 2, b → 3
- ✗ `int [] [] a, [] b;` → CE:

Special

(Conclusion) If you want to declare dimension before the variable, this rule will be applicable only for first variable in a declaration. (of 2-D array). not for second variable. it

If we are trying to apply for next/onwards remaining variables we will get compile time error.

```
int[]    []a, []b, []c;
```



This rule is applicable only for before the variable but will fine for after the variable.

* 3-D Array declaration :-

```
int [][][]    a;
int [    [][][]a;
int      a[][][];
int[]    [][]a;
int[]      a[][];
int[]     []a[];
int[][]     []a;
int[][]      a[];
int        [][]a[];
int        []a[][];
```

all declaration is valid.

2 conclusion :-

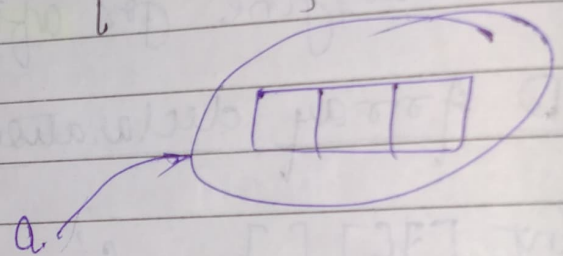
1. At the time of declaration we can't specify the size of an array.
2. To represent dimension before the variable this rule is applicable only for 1 var. not for 2nd var.

* Array Creation :-

Every array in java is an object only. Hence we can create arrays by using new operator.

ex: int[] a = new int[3];

- int[] → D.T
- a → reference variable / array name
- new → keyword
- [3] → size of array
- dimension / array symbol
- use for creating object.



* Usually we can create object only for classes so, that to every array type corresponding classes will be there.

* Every array type correspond. classes are there but these classes are only available at language level not for programmer level.

To see the class :-

sop(a.getClass().getName());

[I → class name for int type var. array.

CLASSTIME Pg. No.
Date / /

Array Type	Corresponding class name
int[]	[I
int[][]	[[I
double[]	[D
short[]	[S
byte[]	[B
boolean[]	[Z

★ Loophole related to array creation :-

1. At the time of array creation compulsory we should specify the size, otherwise we will get C.T.E.

eg:- int[] x = new int[]; ✗
int[] x = new int[6];

int[] x = new int[0];

2. It is legal to have an array with size '0' in java.

ex:- int[] x = new int[0];

3. If we are trying to specify array size with some -ve int value then we will get one runtime exception saying NegativeArraySizeException.

int [] x = new int [-3];

RE: NegativeArraySizeException

* Because the job of compiler is to check only whether we are specifying some size or not. and it is int value or not. -ve & +ve are not checked by compiler.

* and at runtime JVM received the memory for size when JVM see -ve size then through RE:

To specify array size the allowed datatypes are in Java. i.e byte, short, char, int.

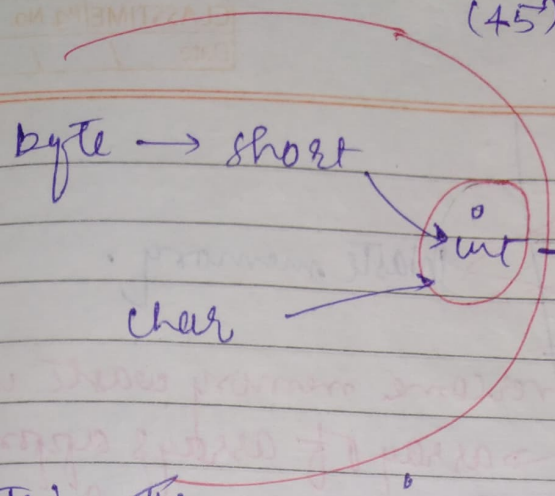
If we are trying to specify any other type then we will get compile time error. Because array.

✓ int [] x = new int [10];
✓ int [] x = new int ['a'];
↳ a → 97 - size of array

byte b = 10;
✓ int [] x = new int [b];

short s = 30;
✓ int [] x = new int [s];

✗ int [] x = new int [10L];
→ CE: PLP found: long req: int.



Note: The maximum allowed array array size in Java is 2147483647 which is the maximum value of int datatype.

1) ✓ int [] x = new int [2147483647];

2) ✗ int [] x = new int [2147483648];
CE: integer number too large.

* Even in the first case we may get runtime exception if sufficient heap memory is not available in our system. like 2147483647 bytes.

↑ upto this above we talk 1D Array creation ↑

Lecture - 7 / 1:12pm / 27-05-22 Sat

2D-Array → also · klas → matrix

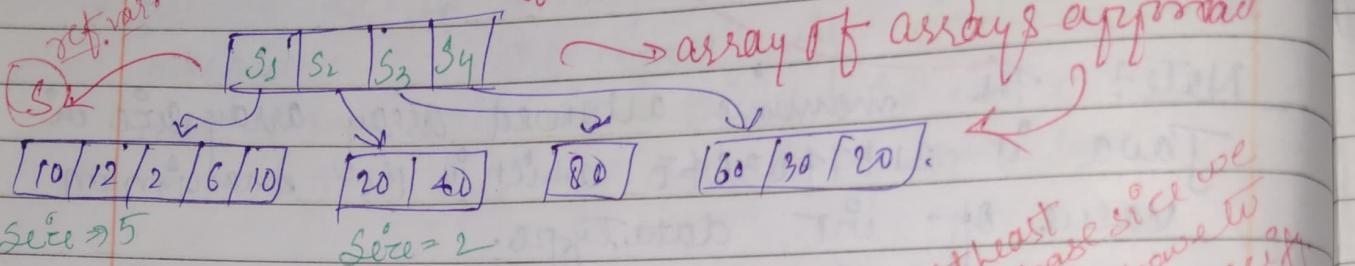
2D-Array Creation: In java 2D-array not implemented by using matrix-style some people followed array of arrays approach for multidimensional array creation.

The main advantage of this approach is memory utilization will be improved.

Student →	S_1	10	12	2	6	10
"	S_2	20	40			
"	S_3	80				
"	S_4	60	30	20		

Waste memory.

To overcome memory waste use array of arrays approach



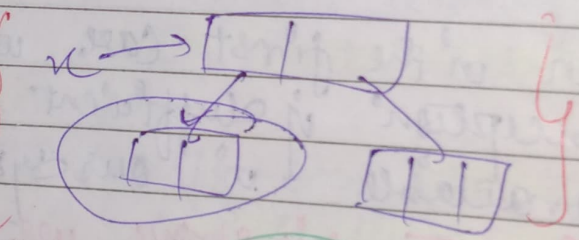
Ex 1 :

```
int [][] x = new int [2][];
x[0] = new int [2];
x[1] = new int [3];
```

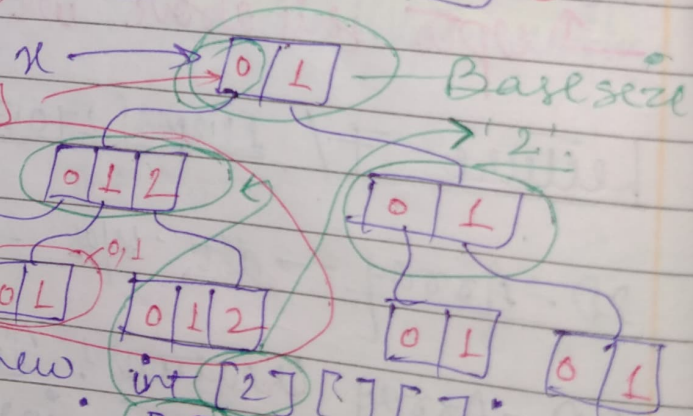
at least base size we have to specify

Corresponding Java code

Memory structure of 2-D array



Ex → 2




```
int [][][] x = new int [2][][];
x[0] = new int [3][];
x[0][0] = new int [1];
x[0][1] = new int [2];
x[0][2] = new int [3];
x[1] = new int [2][2];
```

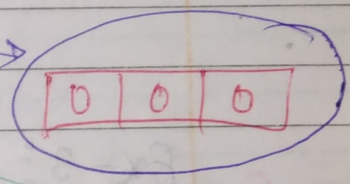
CLASSTIME Pg. No.
Date / /

Which of the following array declarations are valid

1. int [] a = new int [-]; ✗ bcz at least base size should be +ve
2. int [] a = new int [3]; ✓
3. int [][] a = new int [][]; ✗
4. int [][] a = new int [3][]; ✓
5. int [][] a = new int [][4]; ✗
6. int [][] a = new int [3][4]; ✓
7. int [][][] a = new int [3][4][5]; ✓
8. int [][][] a = new int [3][4][]; ✓
9. int [][][] a = new int [3][][5]; ✗
10. int [][][] a = new int [][4][5]; ✗

Array initialization :-

int [] x = new int [3];
ref. var.



sopln (x); → [I@8d00c6
class name. hash code in Hexadecimal form.

sopln (x[0]); → 0

Once we creates an array every array element by default initialized with default values i.e 0.

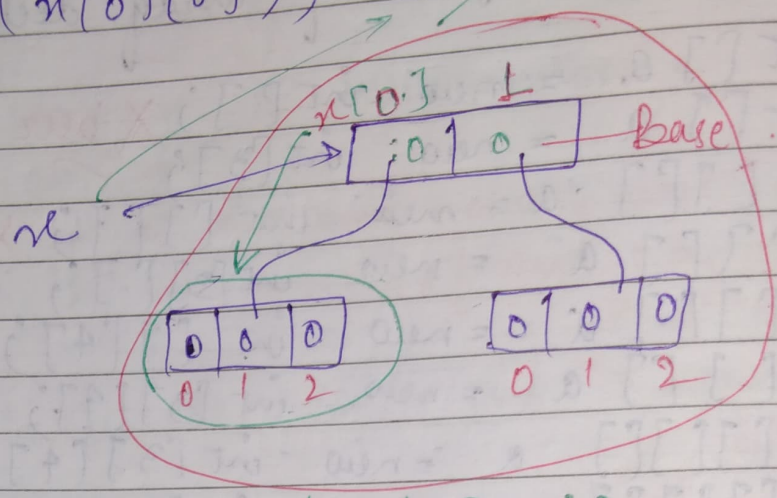
Note :- Whenever we are trying to print any reference variable (x) internally toString() method will be called which is implemented by default to return the String in the following form ⇒

⇒ classname @ hashcode - in - Hexadecimal form.

Ex 2 :- int [][] x = new int [2][3];
sopln (x);
sopln (x[0]);

sopln(x[0][0]);

x is ref var. of 2-dimensional array



→ x[0] is ref var. of 1-D array.
→ x is ref var. of 2-D array.

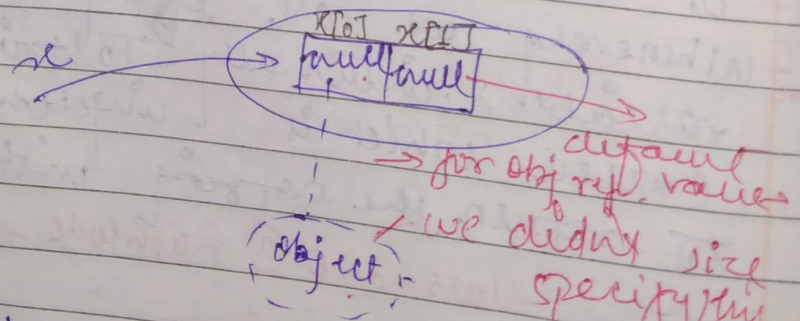
So,

o/p :- [[I @ 3e25a5
[I @ 18e862c
0

Ex-3:-

int[][] x=new int[2][];

sopln(x); → [[I@ 3e25a5
sopln(x[0]); → null
sopln(x[0][0]); → RE: NPE



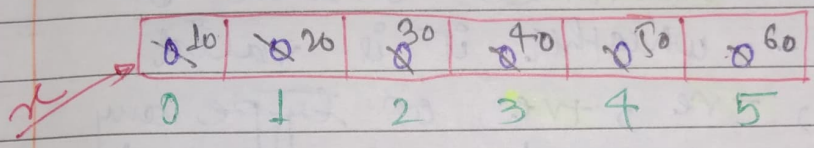
When we didn't specify size it will pointing to 1-D Object. But we didn't specify this, then the

Have doubt

O/P:- [[I@12dcb8c
null
NullPointerException.

Note:- If we are trying to perform any operation on null then we will get (NPE) RuntimeException saying NullPointerException.

⇒ int[] x = new int[6];



its length always not index.

index ⇒ Arraylength − 1
5 ⇒ 6 − 1

x[0] = 10;
x[1] = 20;
x[2] = 30;
x[3] = 40;
x[4] = 50;
x[5] = 60;
x[6] = 70; RE: ArrayIndexOutOfBoundsException.

1. Once we create an array every array element by default initialized with default values. i.e (0).
If we are not satisfied with default values then we can override those values with our customized values.

Because array range is from 0 to 5 + 6

x[-6] = 80; → RE: ArrayIndexOutOfBoundsException.
x[2.5] = 90; → CE: PLP
found: double
req: int

Date / /

Note: by MP

1. Compiler only going to check syntactical mistake. like [in array — wheather it is valid or value or not. of type of array is int.
and otherwise it produce error when found called! Compile time Error/ exception.

2. But JVM is going to check logical mistake at runtime. or like in array ex- wheather it is valid in range, -ve, +ve, or ~~type~~ any other mistake. and when found error called: Runtime Exception.

Note by Sir: If we are trying to access array element without of range index (either +ve value or -ve int value). then we will get RE: ArrayIndexOutOfBounds Exception.

01-06-22 Lecture 8:

* Array declaration, Creation, initialization in a single line:

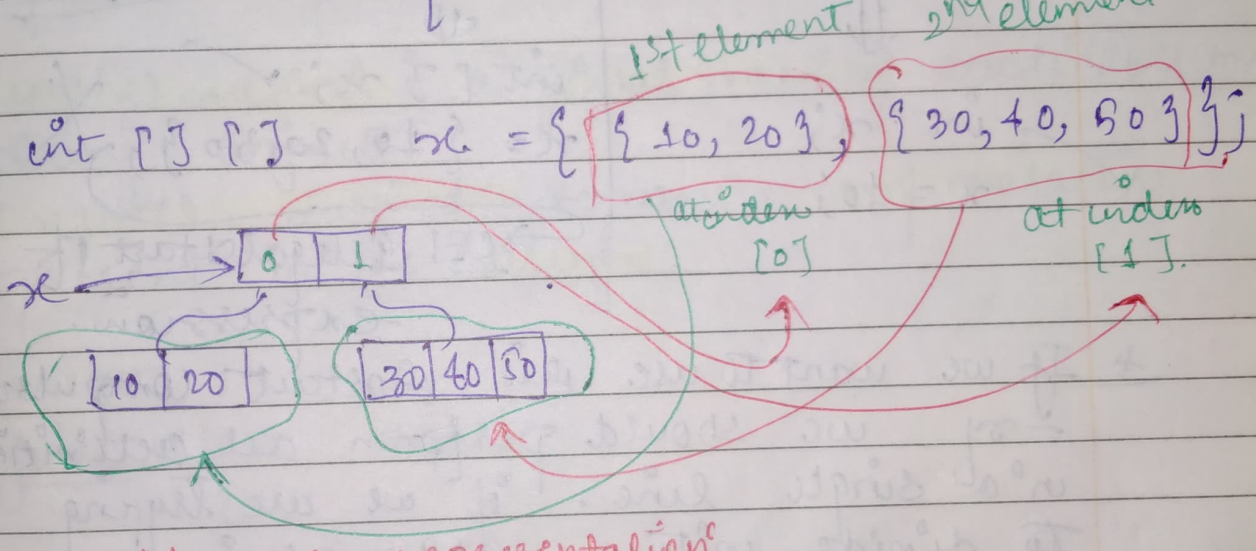
declare: int [] x;
create: x = new int [3];
initialize: x[0] = 10; x[1] = 20; x[2] = 30;

int [] x = {10, 20, 30};
char [] x = {'a', 'e', 'i', 'o', 'u'};
String [] s = {"A", "AA", "AAB"};

We can declare, create, initialize an array in a single line. (shortcut representation).

No problem in performance.
Every Java object will be created in heap only.

Conclusion:-
We can use/extend this shortcut for multidimensional arrays also.



Memory representation

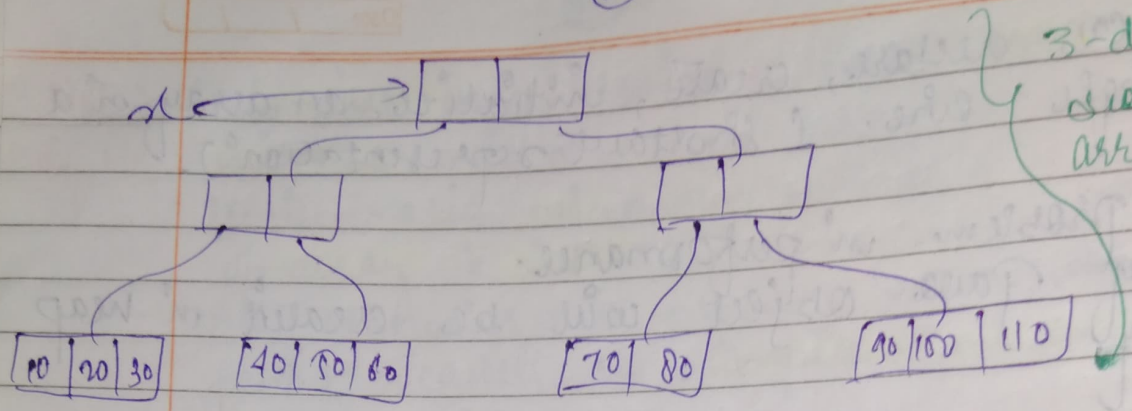
3-D Array Example:-

3 bracket - 3 dimension array.

int [][][] x = {{{10, 20, 30}, {40, 50, 60}}, {{70, 80}, {90, 100, 110}}};

sop(x[0][1][2]); 60
sop(x[1][0][1]); 80
sop(x[2][0][0]); RE: AIOOBE
sop(x[2][1][0]); RE: AIOOBE
sop(x[1][1][1]); 100

sop(x[2][1][0]); RE: AIOOBE



3-dimension array.

02/June/22
Thursday
11:39AM

```
int x = 10;
```
⇓
```
int x;
x = 10;
```
✓

```
int[] x = {10,20,30};
```
⇓
```
int[] x;          ✓
x = {10,20,30};   ✗
```
→ CE: Illegal start of expression.

* If we want to use this shortcut compulsory we should perform all activities in a single line. If we are trying to divide into multiple lines then we will get CTE.

* length vs length() method.

length:-

```
int[] x = new int[6];
```
sopln(x.length()); → CE: cannot find symbol
symbol: method length()
location: class int[]

sopln(x.length); 6

* length is a final variable applicable for arrays.
* length variable represents the size of the array.

length() :-

```
String s = "durga";
sopln(s.length); → CE: Can not find symbol
                     symbol: variable length
                     location: class j.l.String.
sopln(s.length()); → 5
```

length() method is a final method applicable for String objects.
length() method returns no. of characters present in the String.

Concept

```
final class String
{
    length()
    {
    }
}
```

Now, length() present in String class and this class is final. so it's not possible to create child. and also can't override this method because we can override only in child class. that's why this method is by default always final.

Conclusion: Every method present in final class is always final by default.

→ Will see detail in OOPs.

Note: length variable applicable for arrays but not for String objects. whereas length() method applicable for String objects but not for arrays.

Ques: String[] s = {"A", "AA", "AAA"};

① sopln(s.length); ✓

② sopln(s.length()); → CE: Cannot find symbol. symbol: method length() location: class String[]

③ sopln(s[0].length); ✓

④ sopln(s[0].length()); → CE: cannot find symbol. symbol: ~~method~~ variable length. location: class java.lang.String

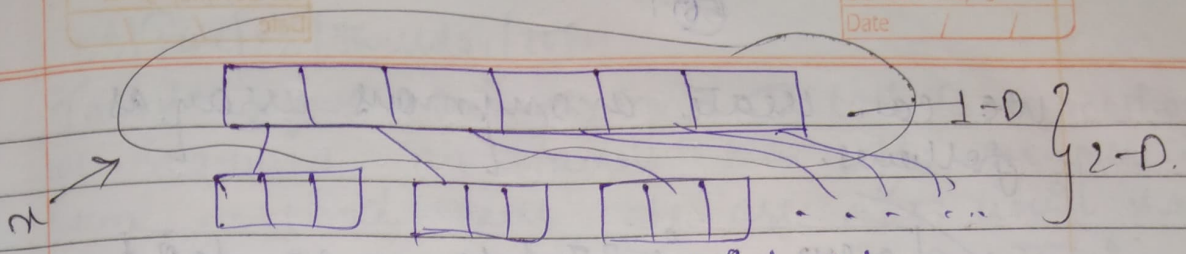
08-Jun-22
1:16 PM

int[][] x = new int[6][3];

In Multidimensional arrays length var. represents only base size but not total size.

sopln(x.length); 6

sopln(x[0].length); 3



There is no direct way to specify length of multidimensional array. but indirectly we can find as follows.

find total

x[0].length + x[1].length + x[2].length + x[2].....

Anonymous Arrays: Sometimes we can declare an array without name. Such type of nameless arrays are called anonymous arrays.
The main purpose of anonymous arrays is just for instant use (one-time usage).

```
class Test
{
    p s v main(String[] args)
    {
        Sum(new int[]{10,20,30,40});
    }
    p s v Sum(int[] x)
    {
        int total=0;
        for(int n:x)
        {
            total = total+n;
        }
        Sop("The Sum"+total);
    }
}
```

We can create anonymous array as follows.

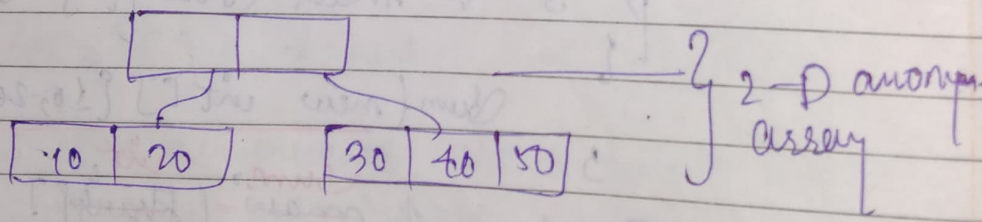
```
new int[]{10,20,30,40};
```

While creating anonymous arrays we can't specify the size. Otherwise we will get C.T.E.

```
new int[3]{10,20,30};   ✗
new int[]{10,20,30};    ✓
```

We can create multidimensional anonymous arrays also.

```
new int[][]{{10,20},{30,40,50}};
```



Based on our requirement we can give the name for anonymous array then it is no longer anonymous.

ex: `int[] x = new int[]{10,20,30};`

name

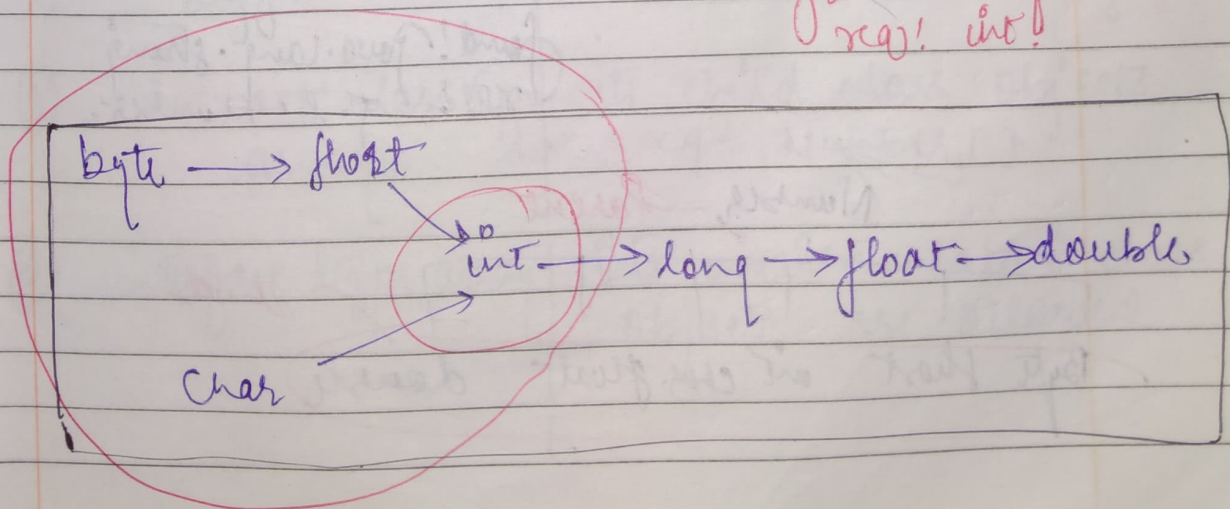
09/June/22/Thursday/1:PM

In the above example just to call Sum method we required an array but after completing sum method call we are not using that array anymore, hence for this one time requirement anonymous array is the best choice.

Array element assignments:

Case 1: In the case of primitive types arrays as array elements we can provide any type which can be implicitly promoted to declared type.

ex:
```
int [] x = new int[5];
x[0] = 10;
x[1] = 'a';
byte b = 20;
x[2] = b;
short s = 30;
x[3] = s;
x[4] = 10L; → CE: PTP found: long req: int
```



ex2:-

Case1:- In the case of float type arrays the allowed D.T are byte, short, int, char, long, float.

Case2:- In the case of object type arrays as array elements we can provide either declared type object or its child class objects.

ex1:-

```
Object[] a = new Object[10];
a[0] = new Object();
a[1] = new String("durga");
a[2] = new Integer(10);
```

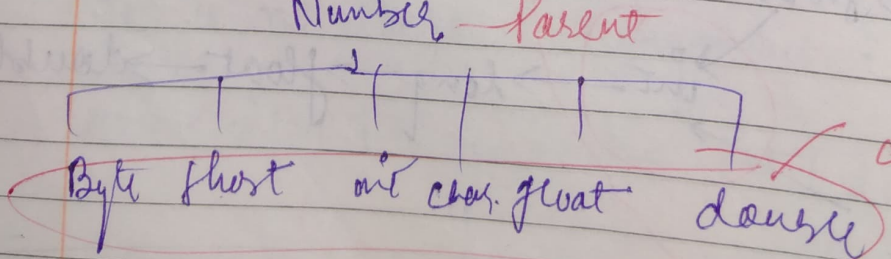
(Object[] — Parent; Integer — child)

ex2:-

```
Number[] n = new Number[10];
n[0] = new Integer(10);
n[1] = new Double(10.5);
n[2] = new String("durga");
```

(Number[] — Parent; Integer, Double — child)

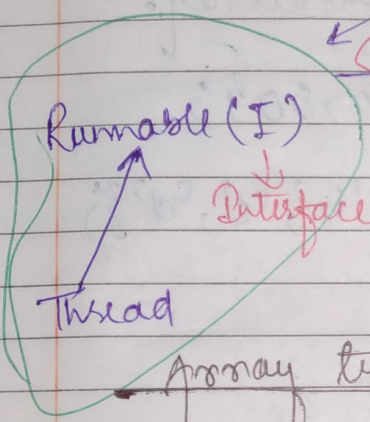
CE: Incompatible types
found: java.lang.String
req: j.l.Number



Cases: for interface type arrays as array elements its implementation class objects are allowed.

```
Runnable [] r = new Runnable [10];
r[0] = new Thread();
r[1] = new String("durga");
```

CE: incompatible types
found: r.l. String
req: j.l. Runnable.



Array type	Allowed element Types.
(1) primitive arrays.	Any type which can be implicitly promoted to declared type.
(2) Object type arrays.	Either declared type or its child class objects.
(3) Abstract class type arrays.	its child class objects. eg: Number[]
(4) interface type array	Its implementation class objects are allowed.

CLASSTIME Pg. No.
Date / /

Lecture - 9 6/09/22 - Thursday - 5:53 PM

Array Variable assignments:

Case1: Element level promotions are not applicable at array level. for eg: char element can be promoted to int type, whereas char[] can not be promoted to int[] array.

eg: int [] x = {10, 20, 30};

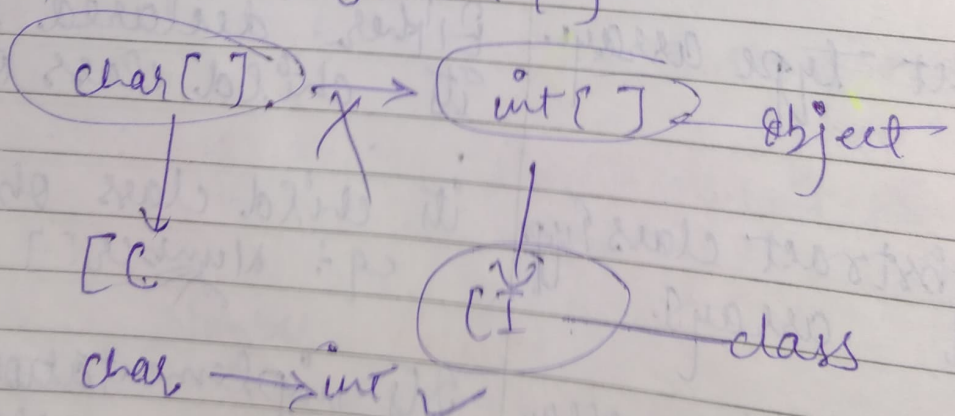
char [] ch = {'a', 'b', 'c', 'd'};

✓ int [] b = x;

✗ int [] c = ch;

CE: Incompatible types
found: char []
req: int []

Reason



CLASSTIME Pg. No.
Date / /

Q. Which of the following promotions will be performed automatically?

- char → int[]
- ~~char[] → int[]~~
- int → double
- ~~int[] → double[]~~
- ~~float → int~~
- ~~float[] → int[]~~
- String → Object
- String[] → Object[].

Reason

→ But in the case of object type arrays, child class type array can be promoted to parent class type array.

child

eg: String[] s = {"A", "B", "C"};

parent

Object[] a = s;

child

parent

Case 2:

Whenever we are assigning one array to another array, internal elements won't be copied. Just reference variables will be reassigned. But condition demands both var have same type.

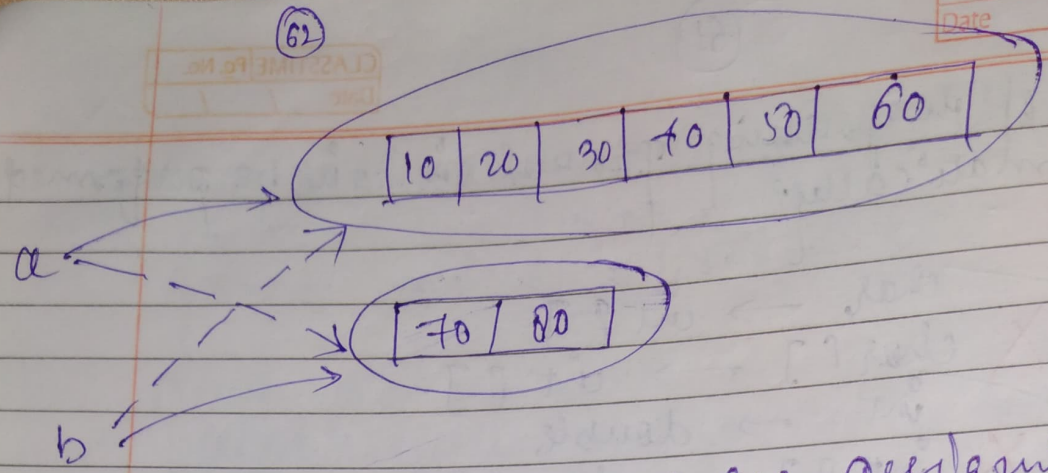
eg:

int[] a = {10, 20, 30, 40, 50, 60};

int[] b = {70, 80};

1. a = b ✓

2. b = a ✓



Case 3: Whenever we are assigning one array to another array ~~int and~~ of dimensions must be matched.

for eg: In the place of 1-D int array we should provide 1-D array only. If we are trying to provide any other dimensions then we will get C.T.E

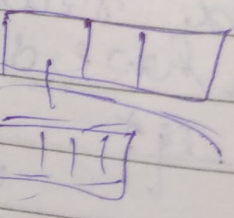
```
int [][] a = new int[5][];

a[0] = new int[4][3];
```

→ CE: incompatible types
found: int[][]
req: int[]

```
a[0] = 10;
```

CE: incompatible types
found: int
req: int[]




```
a[0] = new int[2];
```

CLASSTIME Pg. No.
Date / /

Note:- Whenever we are assigning one array to another array both dimensions and size must be matched, but sizes are not required to match.

Ex 1:

```
class Test
{
    p s v main(String[] args)
    {
        for(int i=0; i<=args.length; i++)
        {
            sopln(args[i])
        }
    }
}
```

java Test A B C ↵

```
A
B
C
AIOOBE
```

java Test A B ↵

```
A
B
AIOOBE
```

java Test

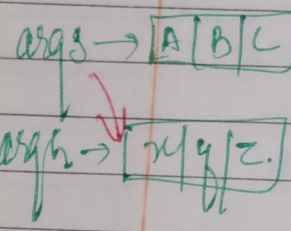
```
AIOOBE
```

```
for (int i=0; i<=3; i++)
{
    sopln (args[i]);
}
```

args[0] args[3] ✗
args[1]
args[2]

eg:-

```
class Test
{
    P S V main (String[] args)
    {
        String[] argh = {"x", "y", "z"};
        args = argh;
        for (String s: args)
        {
            sopln(s);
        }
    }
}
```



Java Test A B C ↵
x, y, z

Java Test A B ↵
xyz

Java Test.
x
y
z.

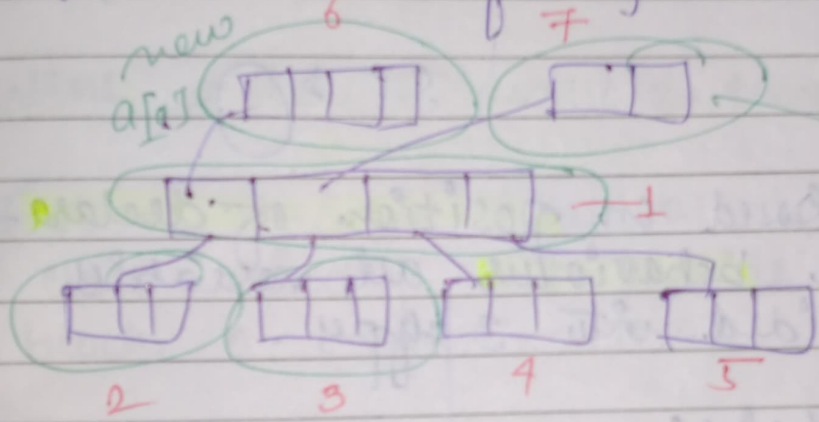
CLASSTIME Pg. No.
Date / /

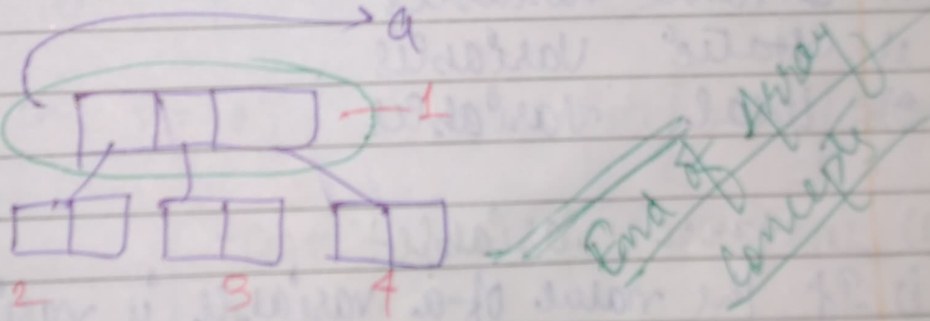
Sol:- int[][] a = new int[4][3]; → 5 objects
a[0] = new int[4]; → 1 obj
a[1] = new int[2]; → 1 obj

a = new int[3][2]; → 4

(1) Total how many objects created? 11

(2) Total how many objects eligible for GC? 7





Lecture 10:- 6/9/2022 — 7:41 PM

* Types of Variables:-

Division 1:-

Based on type of value represented by a variable all variables are divided into 2 types.

1. primitive variables :- Can be used to represent primitive values.

int x = 10;

2. Reference variable :- Can be used to refer objects.

Student S = new Student();

S → O

Division 2 :-

Based on position of declaration and behaviour all variables are divided into 3 types.

1) Instance Variables
2) Static Variables
3) Local Variables

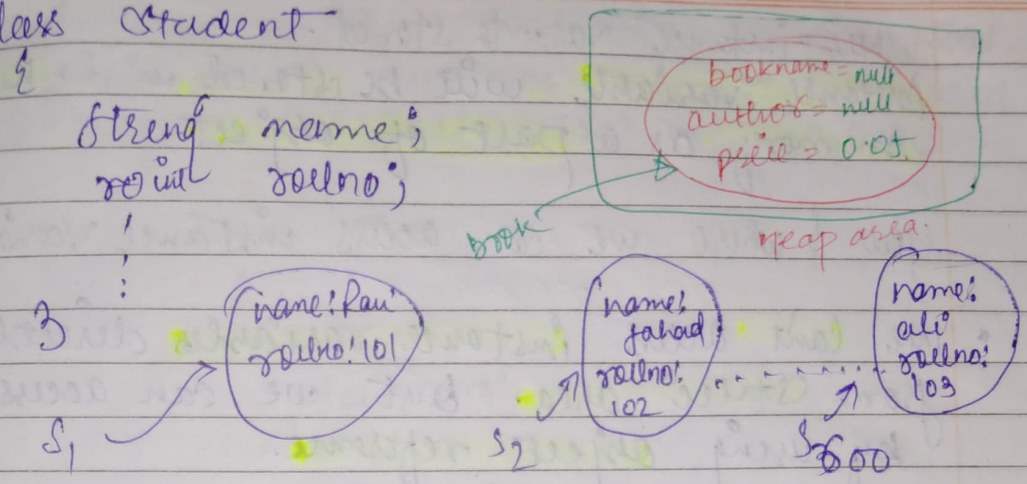
(1) Instance Variable :-

i) If the value of a variable is varied from object to object such type of variables are called instance variables.

2) for every object a seperate copy of instance variables will be created.

CLASSTIME Pg. No.
Date / /

```
class Student
{
    String name;
    int rollno;
    :
    :
}
```



Q Where we have to declare instance variable?

Instance variable should be declared within the class directly but outside of any method block or constructor.

```
class Test
{
    int x=10;
}
```
→ method / block / constructor

```
m1()
{
    int x=10;
}
```

```
Test()
{
    int x=20;
}
```

```
static
{
    int x=10;
}
```

When instance variable will be created..?

instance variable will be created at the time of object creation and destroyed at the time of object destruction, Hence the scope of instance variable is exactly same as the scope of object.

where instance var. is stored.

Instance variable will be stored in the heap memory as a part of object.

How & where we can access instance variable.

- We can't access instance variables directly from static area. but we can access by using object reference.
- But we can access instance variables directly from instance area.

eg:-

```
class Test
{
    int x=10;
    p s v main(String[] args)
    {
        sopln(x);   --> C.E: Non-static variable x cannot be referenced from a static content
        Test t=new Test();
        sopln(t.x); 10
    }
    public void m1()
    {
        sopln(x);
    }
}
```

instance var.

static area.

Instance area.

access by using object reference

* Instance variables also known as object level variables or attributes.

CLASSTIME Pg. No.
Date / /

* for instance variable JVM will always provide default values. and we are not required to perform initialization explicitly.

eg:
```
class Test
{
    int x;
    double d;
    boolean b;
    String s;
    public static void main(String[] args)
    {
        Test t1 = new Test();
        sopln(t1.x);    0
        sopln(t1.d);    0.0
        sopln(t1.b);    false
        sopln(t1.s);    null.
    }
}
```

* Static variables:

class Student.

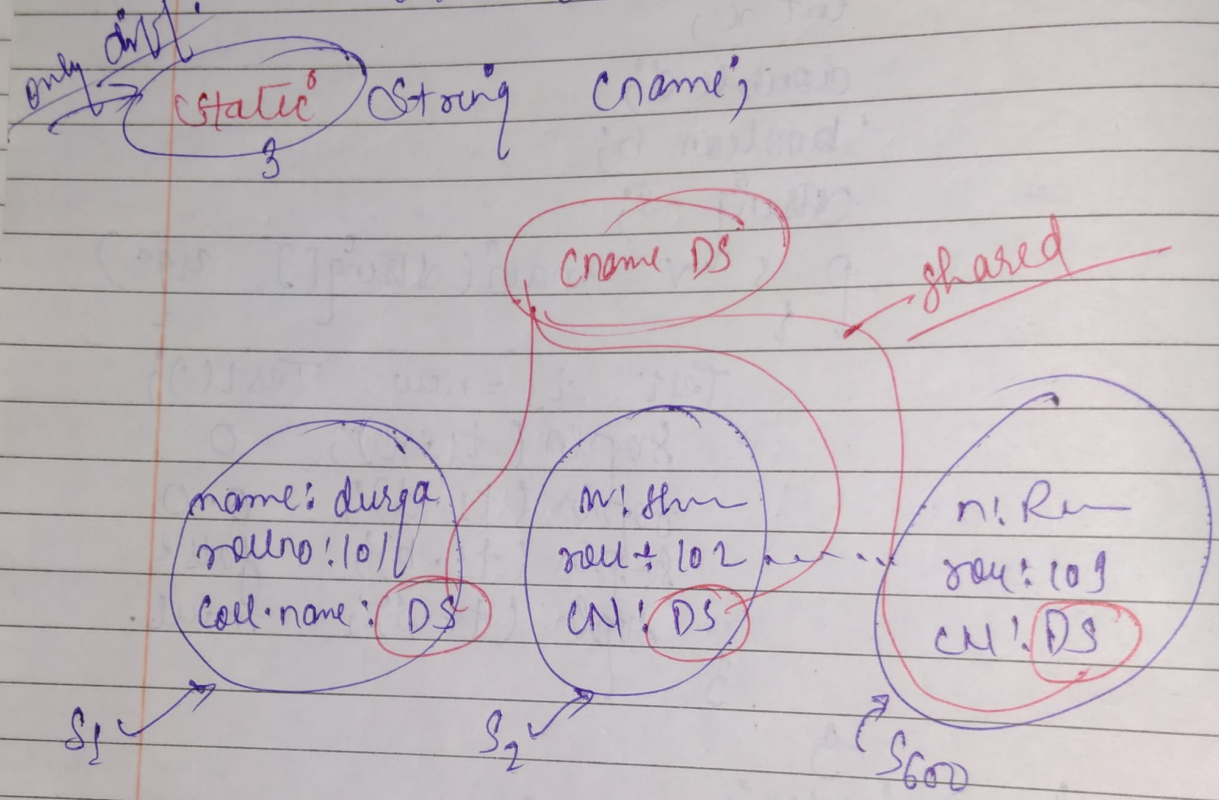
If the value of a variable is not varied object to object then it is not recommended to declare variable as instance variable. we have to declare such type of variable at class level by using static modifier.

In the case of instance variables for every object a seperate copy will be created but in the case of static variables a single copy will be created at class level and shared by

every object of the class.

```
class Student
{
    String name;
    int rollno;
    static String cname;
}
```

only one



Where we declare static variables:
static var. should be declared within the class directly but outside of any method, block or constructors.

When static var. created & destroyed.

static variables will be created at the time of class loading and destroyed at the time of class unloading. Hence scope of static variable is exactly same as

CLASSTIME Pg. No. Date / /

scope of that class file.

Java Test

1) Start JVM
2) Create & start main Thread (by JVM)
3) Locate Test.class file (by main Thread).
4) Load Test.class → (static var. creation) (By M.T)
5) Execute main() method. → (By Main Thread)
6) Unload Test.class (by main Thread). → static var. destroyed.
7) Terminate main Thread (by JVM)
8) Shut down JVM

Error: Exception in thread main (no class def found err)
↳ of Test.class file not found. ↑ step - 3

static variable will be stored in method area (memory).

How we can access static variable:
We can access static variables either by object reference or by class name but recommended to use class name.
Within the same class it is not required to use class name and we can access directly.

```
class Test
{
    static int x = 10;
    p s v main(String[] args)
    {
```

```
{
    Test t = new Test();
                  → object
3-ways to access static var.
    1) sopln(t.x);
    2) sopln(Test.x);   → recommended
    3) sopln(x);        → class.
                        → directly but within the same class.
}
```

* We can access static variables directly from both instance and static areas.

```
class Test
{
    static int x = 10;
    public static void main(String[] args)
    {
        sopln(x);
    }
    public void m1()
    {
        sopln(x);
    }
}
```

static area.

instance area

When no any initialization for static variable.

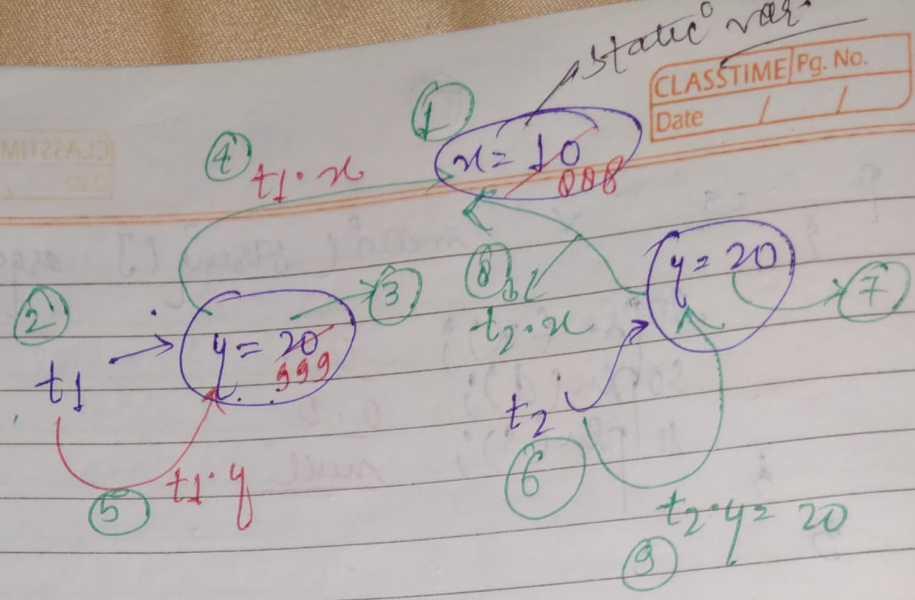
```
class Test
{
    static int x;
    static double d;
    static String s;
```

```
p  s  v  main(String[] args)
{
    sopln(x);   0
    sopln(d);   0.0
    sopln(s);   null
}
}
```

→ For static variables JVM will provide default values and we are not required to perform initialization explicitly.

→ static variables also known as class level variables or Fields.

```
class Test
{
    static int x=10;
    int y=20;
p   s   v   main(String[] args)
{
    Test t1 = new Test();
    t1.x = 888;
    t1.y = 999;
    Test t2 = new Test();
    sopln(t2.x + "....." + t2.y);
}                  888 ~~~~ 20
}
```



Lecture 11 : 8/10/22 - 4:37 PM.

* Local Variables :- / Temporary variable / Stack variable / Automatic variables.

1. Sometimes to meet temporary requirements of the programmers we can declare variables inside a method or block or constructor. Such type of variables are called local variables or Tempory Var. or stack, automatic variables.

2. Local variables will be stored inside stack memory. so also called stack variable.

main method	Block	Constructor

main1()
{
int x=10;
}
	static
	{
	int x=10;
	}
		Test()
		{
		int x=20;
		}

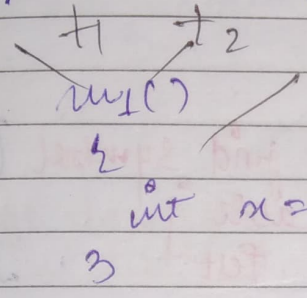
for(int i=0; i<=3; i++)
{
sysout(i);
}

automatic

#5

When local var. created & Scope :-

Local variables will be created while executing the block in which we declared it. Once block execution completes automatically local var. will be destroyed Hence the scope of local var. is exactly a block in which we declared it.



for every thread a seperate copy of local variable is created.
So, L.V are also called thread safe.

Ex (1)
```
class Test
{
  p s v main(String[] args)
  {
    int i=0;
    for(int j=0; j<3; j++)
    {
      i=i+j;
    }
    sopln(i+"....."+j);
  }
}
```

o/p: Cannot find Symbol
found: variable j
Test: class Test.

(2)
```
class Test
{
  p s v m(String[] args)
  {
    try
    {
```

```
int j = Integer.parseInt("ten");
}
catch (NumberFormatException e)
{
    j = 10;
}
System.out.println(j);
}
```

CE: cannot find symbol
symbol: variable j
location: class Test

For local variables JVM won't provide default values compulsory we should perform initialization explicitly before using that var. i.e. if we are not using then it is not required to perform initialization.

(3)
```
class Test
{
    p s v main(String[] args)
    {
        int x;
        System.out.println("Hello");
    }
}
```

CLASSTIME Pg. No.
Date / /

(4) class Test
{
  p s v main(string[] args)
  {
    int x;
    sop ln(x);
  }
}

CE: Variable x might not have been initialized.

(5) class Test
{
  p s v main(string[] args)
  {
    int x;
    if (args.length > 0)
    {
      x = 10;
    }
    sop ln(x);
  }
}

CE: Variable x might not have been initialized.

(6) class Test
{
  p s v main(String[] args)
  {

```
int x;
if (args.length > 0)
{
    x = 10;
}
else
{
    x = 20;
}
sopln(x);
}
```

O/p : Java Test A B
      O/p : 10

Java Test
O/p : 20.

Note : (1) It is not recommended to perform initialization for local variables inside logical blocks. because there is no gaurantee for the execution of these blocks always at runtime.

(2) It is highly recommended to perform initialization for local variables at the time of declaration. at least with default values.

CLASSTIME Pg. No.
Date / /

The only applicable modifier for local variables is FINAL. By mistake if we are trying to apply any other modifier then we will get CTE.

```
class Test
{
    public int x=10;
    private
    <default>
    protected
    static int y=20;

    p s v main(str[] args)
    {
        final int z=30;
        public
        static
    }
}
```

```
class Test
{
    p s v main(String[] args)
    {
        public int x=10;
        private int x=10;
        protected int x=10;
        static int x=10;
        transient int x=10;
        volatile int x=10;

        final int x=10;
    }
}
```

CE: illegal start of expression.

CLASSTIME Pg. No.
Date / /

Note: If we are not declaring with any modifier then by default it is default. But this rule is applicable only for instance and static variables but not for local variables.

Summary Conclusion abt Instance / Local / static

Conclusions:

(1) For instance and static variables JVM will provide default values and we are not required to perform initialization explicitly. But for local variables JVM won't provide default values Compulsory we should perform initialization explicitly before using that variable.

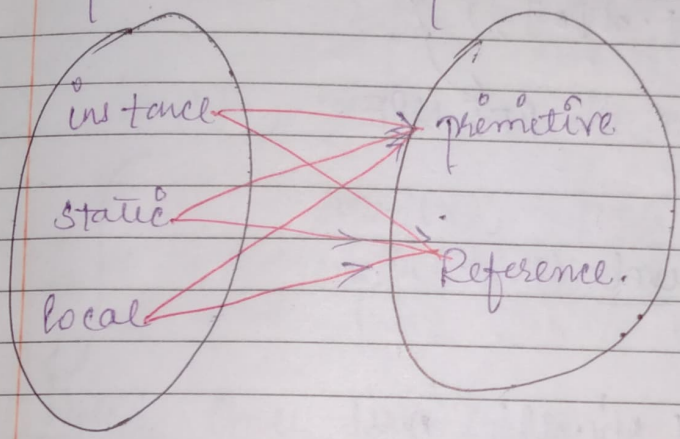
(2) Instance and static variables can be accessed by multiple threads simultaneously and hence these are not thread safe, but local variables case for every thread seperate copy will be created and hence local variables are thread safe.

1) Every variable in Java should be either instance / static or local.

2) Every variable in Java should be either primitive or reference / Non-primitive.

Type of variable	Is Thread-Safe?
1) instance variables	✗
2) static variables	✗
3) local variable	✓

→ Hence various possible combinations of java variables



Ex:-

```
class Test
{
    instance-primitive→int x=10;
    static-reference→static String s="durga";

    P S V main(String[] args)
    {
        local-reference → int[] y=new int[3];
    }
}
```

CLASSTIME Pg. No.
Date / /

* Uninitialized arrays :-

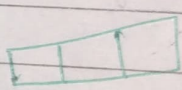
```
class Test
{
    int[] x; // inst-ref
    p s v main(String[] args)
    {
        Test t = new Test();
        sopln(t.x); null
        sopln(t.x[0]); X
    }
}
```

RE: NPE.

(1) (I) Instance Level

(1) `int[] x;`
`sopln(obj.x);` null
`sopln(obj.x[0]);` RE: NPE

(2) `int[] x = new int[3];`



`sopln(obj.x);` [I e3e25a5
`sopln(obj.x[0]);` 0

(2) (II) Static Level :-

(1) `static int[] x;`
`sopln(x);` null
`sopln(x[0]);` RE: NPE

CLASSTIME Pg. No. Date / /

(2) static int [] x = new int [3];

soplu(x); [I @ 3e25a5
soplu(x[0]); 0.

(3) III local Level :-

(1) int [] x;

soplu(x);
soplu(x[0]); } (E! variable x might not have been initialized.

(2) int [] x = new int [3];

soplu(x); [I@3e25a5
soplu(x[0]); 0.

Note :- Once we create an array every array element by default initialized with default values. irrespective of whether it is instance, static or Local array.

x is loc.var. but x.[0] is a part of array object. Array property is going to be dominated.

Lecture 12 :-

Var-arg Methods :- (Variable number of Argument methods) :-

Until 1.4 version we can't declare a method with variable no of arguments if there is a change in no of arguments compulsory we should go for new method. It increases

CLASSTIME Pg. No.
Date / /

Length of the code and reduces readability.

To overcome this problem some people introduced var-arg methods in 1.5 version according to this we can declare a method which can take variable no. of arguments such type of methods are called var-arg methods.

```
m1 (int... x)
      ↳ 3 dots only
```

We can declare a var-arg methods as follows above.

We can call this method by passing any number of int values including zero number.

```
m1 ();  ✓
m1 (10);  ✓
m1 (10,20);  ✓
m1 (10,20,30,40);  ✓
```

Ex:

```
class Test
{
  public static void m1(int...x)
  {
    System.out.println("var-arg method");
  }
  p s v main(String[] args)
  {
```

CLASSTIME Pg. No.
Date / /

```
m1();
m1(10);
m1(10,20);
m1(10,20,30,40);
}
}
```

o/p: var-arg.method(4+4

```
sum(10,20);
sum(10,20,30);
sum(10,20,30,40);
sum(10,20,30,40,50);
```

```
ps v sum(int a, int b)
{ soplu(a+b);
}
p s v sum(int a, int b,
          int c)
{ soplu(a+b+c);
}
p s v sum(int a, int b,
          int c, int d)
{ soplu(a+b+c+d);
}
...
p s v sum(int a,
          int b, int ...)
{ soplu(a+b+c...);
}
```

But this method is replaced from 1.5v of Java onwords.

sum(int... x)

Internally var-arg parameter will be converted into 1-D array, hence with in the var-arg method we can differentiate values by using index.

CLASSTIME Pg. No.
Date / /

```
m1 (int... x)
{
    Sysout
}
```

int [] x
↳ 1-D array.

ext: Let me check | verify.

```
public static void m1 (int..x)
{
    Sopln ("The no of arguments" + x.length);
}
public static void main (String[] args)
{
    m1();
    m1 (10);
    m1 (10,20);
    m1 (10,20, 30,40);
}
```

opt	0
	1
	2
	4

(2) Ex :- (Var-arg method example)

```
class Test
{
    public static void main(String[] args)
    {
        Sum();
        Sum(10,20);
        Sum(10,20,30);
        Sum(10,20,30,40);
    }
    Pub St. void Sum(int... x)
    {
        int total = 0;
```

(Q7)

```
for (int x1 : x)
{
    total = total + x1;
}
System.out.println("The Sum:" + total);
}
```

O/P :-

The Sum: 0
The Sum: 30
The Sum: 60
The Sum: 100

Loophole :-

— Syntactical Kind of Loophole :-

Case 1 :- Which of the following are valid var-arg method declaration ?

m1 (int[] x)	m1 (int... x) ✓
m1 (int [] x)	m1 (int ... x) ✓
m1 (int x[])	m1 (int x...) ✗
	m1 (int ... x) ✓
	.
	m1 (int . ..x) ✗
	m1 (int .x..) ✗

Case 2 :- We can mix var-arg parameter with normal parameter.

m1 (int x, int... y) ✓
m1 (String s, double... y) ✓

Case 3:- If we mix normal parameter with var-arg parameter then var-arg parameter should be last parameter.

```
m1 ( double ... d , String s)   ✗
m1 ( char ch , String ... s)    ✓
```

Case 4:- Inside var-arg method we can take only one var-arg parameter and we can't take more than one var-arg parameter.

```
m1 ( int ... x , double ... d)   ✗
```

Case 5:- Inside a var-arg method class we can't declare var-arg method and the corresponding 1-D array method simultaneously otherwise we will get CTE.

OOPS Concept

```
class Test
{
    ps v m1(int ... x)   ⇒ m1(int[])
    {
        sopln("int ...");
    }
    ps v m1(int[] x)   ⇒ m1(int[])
    {
        sopln("int[]");
    }
}
```

int[]

✗

✗

CF! Cannot declare both m1(int[]) and m1(int...) in Test.

```
class Test                                  t.m1(10);
{
  public void m1(int i)                     m1(int)
  {
  }
  public int m1(int i)                      m1(int)
  {
    return 10;
  }
}
```

OOPS Concept

→ Compiler gets confused which m1 method we call. ← means ↑

→ In a class same signature method we can't call at a time.

Inside a class two method with same signature we can't call.

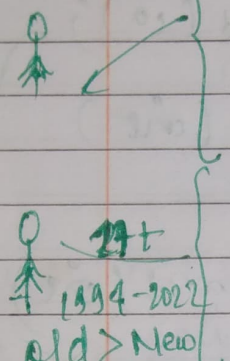
Ex:

```
class Test
{
  p s v m1(int... x)
  p s v m1(int[] x)
}
```

↳ CF! Can't declare both m1(int[]) and m1(int...) in Test.

Case 6:

```
class Test
{
    public static void m1(int ... x)   V.A
    {
        sopln("var-arg method");
    }
    public static void m1(int x)   G.M
    {
        sopln("General method");
    }
    p s v main(String[] args)
    {
        m1();      var-arg method
        m1(10,20); var-arg method
        m1(10);    General method.
    }
```



Same as / such as

```
switch(x)
{
    case 1:
    case 2:
    case 3:
    default:   — less priority
}
```

In general var-arg method will get least priority i.e. if no other method matched then only var-arg method will get the chance it is exactly same as default case

CLASSTIME Pg. No.
Date / /

inside a switch.

Array Concepts :-

Case 1 :- Wherever 1-D array present we can replace with var-arg parameter.

m1 (int[] x) ⇒ m1 (int... x)

ex:-

main (String[] args) ⇒ main (String... args)

→ Extra eg :-

m1 (int[] x)	m1 (int... x)
m1 (new int[] {10});	✓
m1 (new int[] {10,20});	✓
m1 (new int[] {10,20,30});	✓
	m1 ();
	m1 (10);
	m1 (10,20,30);

Equivalence b/w var-arg parameter and 1-D array.

→
```
class Test
{
    public static void main(String... args)
    {
        Sopln("var-arg main");
    }
}
```

Case 2:- Wherever var-arg parameter present we can't replace with 1-D array.

~~m1(int... x) ≡ m1(int[] x)~~

m1(int... x)	m1(int[] x)
m1();	✗
m1(10);	✗
m1(10,20,30);	✗
m1(new int[] {10,20});	✓

not array

Dimension:-

1. m1(int... x) we can call this method by passing a group of int values and x will become 1-D array. (int[]x)

2. m1(int[]... x) we can call this method by passing a group of 1-D int arrays and x will become 2-D int array. (int[][] x.).

1. m1(int... x) ≡ int[] x
2. m1(String... x) ≡ String[] x

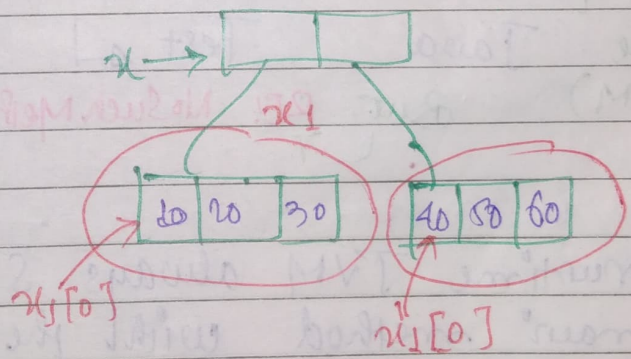
CLASSTIME Pg. No.
Date / /

3. m1 (int []... x) ≡ int [][] x
4. m1 (int[][][]... x) ≡ int [][][] x.

⇒
```
class Test
{
    p s v main (String[] args)
    {
        int [] a = {10,20,30};
        int [] b = {40,50,60};
        m1 (a,b);
    }
public static void m1 (int[]... x)
    {
        for (int[] x1 : x)
        {
            sopln (x1[0]);
        }
    }
}
```

also

o/p: 10
40.



2-D array

Lecture 13 :- (Main Method) :- 06/10/22 : 1:14PM

1. Whether class contains main method or not and whether main method is declared according to requirement or not these things won't be checked by compiler. at runtime JVM is responsible To check these things. if JVM unable to find main method then we will get Runtime exception saying:

RE: NoSuchMethodError : main.

Ex: class Test { }

1) Compilation Javac (compiler) Test.Java ✓ Compiled Successfully.

2) Runtime Java (JVM) Test ↵
But RE: NoSuchMethodError : main.

2:- At runtime JVM always searches for the main method with the following prototype.

public static void main (String[] args)

CLASSTIME Pg. No.
Date / /

public static void main(String[] args)

- public: To call by JVM from anywhere
- static: without existing Object also JVM has to call this method.
- void: main() method won't return anything to JVM.
- main: This is the name which is configured inside JVM.
- (String[] args): Command-Line arguments.

The above system is very strict and if we perform any change then we will get Runtime Exception: NoSuchMethodError: main.

Valid

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("valid main");
    }
}
```

Even though above syntax is very strict the following changes are acceptable.

(1) static public :- Instead of public static we can take static public. i.e. the order of modifiers is not important.

(2) We can declare String array in any acceptable form.

ex:
```
main(String[] args)      ✓
main(String args[])      ✓
main(String... args)     ✓
```

(3) Instead of args we can take any valid Java identifiers.

ex:
```
main(String[] durga).
main(String[] anyname).
```

(4) We can replace String[] with var-arg parameter.

ex: main(String... args).

(5) We can declare main method with the following modifiers :-

1) final
2) synchronized
3) strictfp.

ex:
```
class Test
{
    static final synchronized strictfp public void main
    (String... durga)
    {
        System.out.println("valid main method");
    }
}
```

Que: Which of the following main method declarations are valid :- ?

1) public static void main (String args) → [] miss ✗
2) public static void Main (String [] args) ✗
3) public void main (String[] args) → ✗ - static miss
4) public static int main (String [] args) → ✗
5) final synchronized strictfp public void main (String [] args) → ✗ - static miss
6) final synchronized strictfp public static void main (String [] args) → ✓
7) public static void main (String... args). ✓

Que2: In which of the above cases we will get Compile time error.

Ans: We won't get CTE anywhere, but except last two cases in remaing we will get RTE saying: NoSuchMethodError : main.

Things related to Syntax of main method.

Functionality wise :-

Case1: Overloading of the main method is possible, but JVM will always call String[] array args main method only. the other overloaded method we have to call explicitly like normal method call.

Ex: →

```
class Test
{
    p s v main (String[] args)
    {
        op1n ("String[]");
    }
    p s v main (int[] args)
    {
        sop1n ("int[]");
    }
}
```

Overloaded Method.

op:- String[]

⇒ Two method having same name but different arguments typed called — method overloading

⇒ Overloading of main method of is also possible.

1)

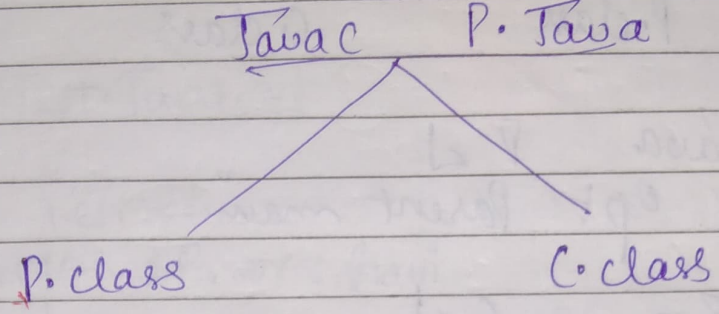
Case 2:- Inheritance concept applicable for main method. Hence if while executing child class if child doesn't contain main method then parent class main method will be executed

2)

ex:- P.Java

```
class P
{
    p s v main(String[] args)
    {
        sop1n ("parent main");
    }
}
class C extends P
```

{ 4
3



Java P ↵
O/p :- Parent main

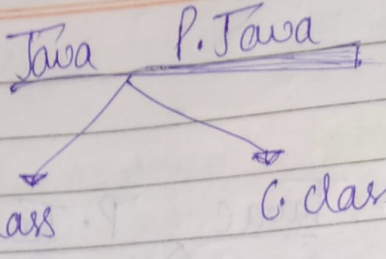
Java C ↵
O/p :- Parent main.

Case3 :- class P
{
P s v main(String[] args)
{
sopln("parent main");
}
}

It is method hiding but not overriding

class C extends P
{
P s v main(String[] args)
{
sopln("child main");
}
}

CLASSTIME Pg. No.
Date / /



Java P ↵
Op:- Parent main

Java C ↵
Op:- child main

→

→ It seems overriding concept applicable for main method but it is not overriding and it is method hiding.

Note:- For main method inheritance and overloading concepts are applicable but overreding concept is not applicable. instead of overriding method hiding is applicable.

1)

2)

* Lecture 14:- 6/19/2022 - 2:37PM (1.7version onwords).

* 1.7v Enhancement wrt main method:-

Until 1.6v if the class doesn't Contain main method then we will get RTE Say: NoSuchMethodError : main.
But from 1.7v Onwords instead of NoSuchMethodError we will get more elaborated error information.

Ex: class Test
{
}

1.6 ✓
Javac Test.java ↵

Java Test ↵
NoSuchMethodError: main.

1.7 ✓
Javac Test.java ↵
Java Test ↵

Error! Main method not found in class Test, please define the main method as:

public static void main(String[] args).

(2) * From 1.7V onwards main method is mandatory to start program execution. Hence even though class contains static main method block it won't be executed if the class doesn't contain main method.

Ex:
```
class Test
{
    static
    {
        System.out.println("Static Block");
    }
}
```

CLASSTIME Pg. No.
Date / /

1.6V	1.7V
Javac Test.Java	Javac Test.Java
Java Test ↵	Java Test ↵
Op: static Block RE: NoSuchError Method: main	Error: Main method not found in class Test, please define the main method as: public static void main(String[] args).

Ex2:

```
class Test
{
    static
    {
        System.out.println("static Block");
        System.exit(0);
    }
}
```

1) → should close JVM.

2)

1.6V	1.7V
Javac Test.Java	Javac Test.Java ✓
Java Test ↵	Java Test ↵
Op: static Block	Bug fixed here. Error: Main method not found in class Test, please.....

Bug is here

Ex 3:-
```
class Test
{
    static
    {
        System.out.println("Static Block");
    }
    public static void main(String[] args)
    {
        System.out.println("main method");
    }
}
```

1.6 V	1.7 V
Javac Test.java	Javac Test.java
Java Test	Java Test
O/P :- Static Block main method	O/P :- Static Block main method.

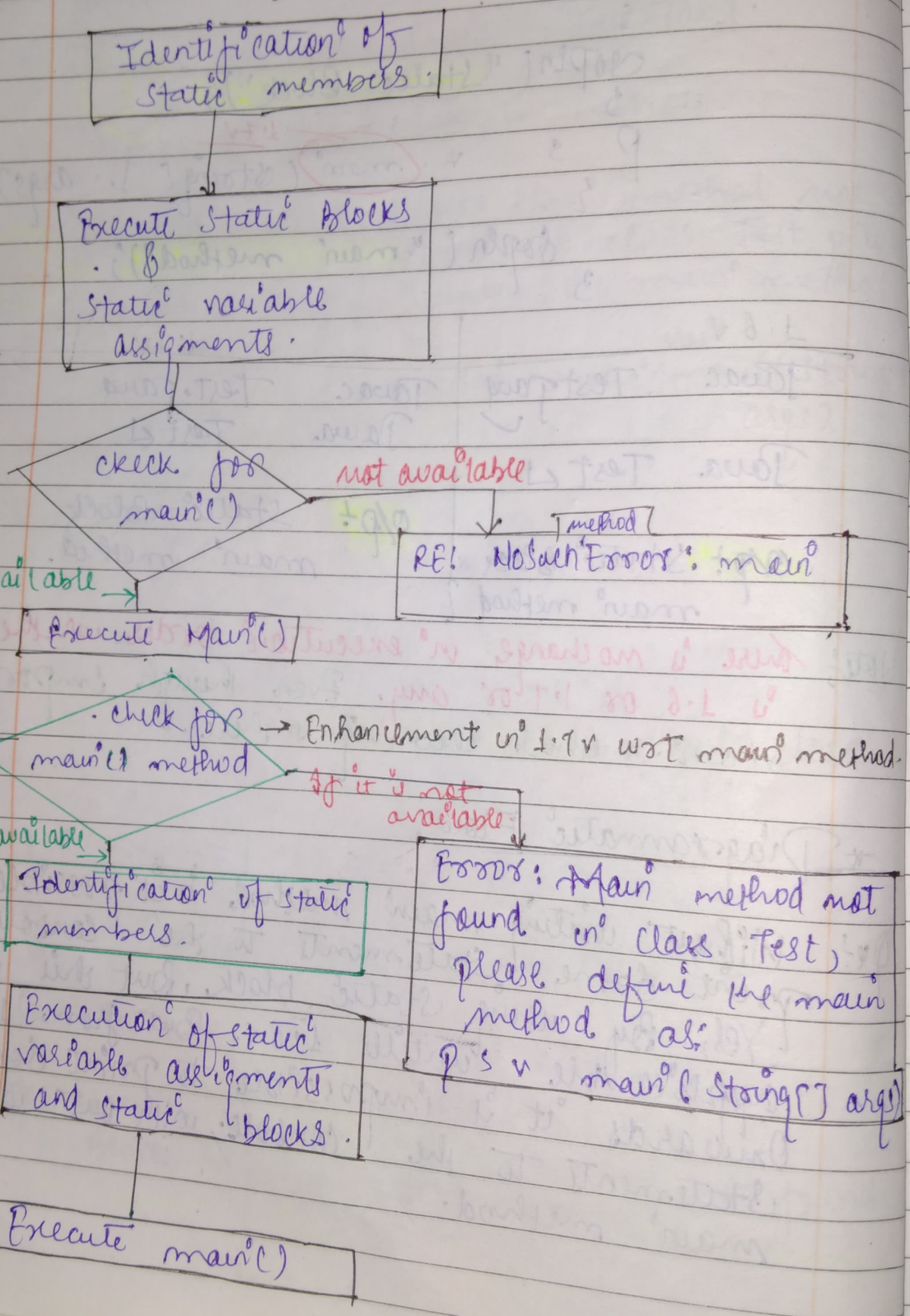
Note:- There is no change in execution order whether version is 1.6 or 1.7 or any. Even though importance of main method checking is there.

* Programatic Flow :-

Qn:- Without writing main method is it possible to print some statements to the console?
Yes, By using static block, But this rule is applicable until 1.6 v. But from 1.7v Onwards it is impossible to print some statements to the console without writing main method.

Date

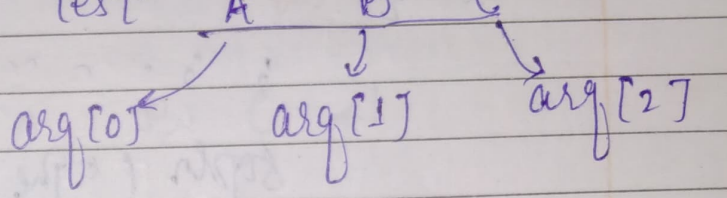
1.6 v flow of execution



5/1/22 Lecture 15 :- (9) Command Line Arguments :- 1:20 PM

The arguments which are passing from command prompt are called CLA. With these CLA JVM will create an array and by passing that array as argument JVM will call main method.

Ex:- Java Test A B C



args.length ⇒ 3.

String[] args = {"A", "B", "C"};

```
p s v main(String[] args)
{
:
:
}
```

Q:- What is the purpose/Need of CLA :-

The main objective of CLA arguments is we can customized behaviour of the main method.

Ex:-

```
class Test
{
    p s v main(String[] args)
    {
        int n = Integer.ParseInt(args[0]);
```

CLASSTIME Pg. No.
Date / /

```
Sopln("The square of " + n + " is: " + (n*n));
}
```

Java Test 4 ↵ → 16
Java Test 5 ↵ → 25.

→
```
class Test
{
    p s v main(String[] args)
    {
        ; ; . - ..
        Sopln("The square of 4 is 16");
    }
}
```
Java Test 4 ↵

→
```
class Test
{
    p s v main(String[] args)
    {
    }
}
```
a.txt → args[0]
b.txt → args[1]
c.txt → args[2]
dont hard code values like this

Que: Why CLA always in String form?
1) The most commonly used object in Java is String type
2) from String to any other type we can able to convert in int, boolean, etc., number or explicitly. etc.

Dry Run

CLASSTIME Pg. No.
Date / /

```
class Test
{
  P  S  V  main (String[] args)
  {
    for (int i=0; i<=args.length ; i++)
    {
      sopln (args[i])
    }
  }
}
```

replace with <

Java Test A B C ↵

```
A
B
C
RE! A100BE
```

Java Test A B ↵

```
A
B
RE! A100BE
```

Java Test ↵

```
RE! A100BE.
```

If we replace <= with < then we wont get any runtime exception.

Dry Run

```
for (int i=0 ; i<=3 ; i++)
{
  sopln (args[i])
}
```

args[0] → A , args[1] → B , args[2] → C args[3] → A100B

CLASSTIME Pg. No.
Date / /

Ex:-
```
class Test
{
    p s v main (String[] args)
    {
        String[] argh = {"x", "y", "z"};
        args = argh;
        for (String s : args)
        {
            System.out.println(s);
        }
    }
}
```

(1)

args → | A | B | C |

But after (1) argh → | X | Y | Z |

2) args = argh. → Now, args → | X | Y | Z |

Java Test A B C ↵

X
Y
Z

Java Test A B ↵

X
Y, Z

Java Test ↵

X Y Z.

CLASSTIME Pg. No.
Date / /

Note: With in main method command line arguments are available in string form.

```
class Test
{
    p s v main(String[] args)
    {
        sopln(args[0] + args[1]);
    }
}
```

String form.

Concatenation if String not arithmetic operation.

Op: 1020

```
Java Test 10 20
```

→ Usually space itself is the seperater b/w CLA. If our CLA itself contains a space then we have to enclose that CLA within double Quotes.

Ex:

```
class Test
{
    p s v main(String[] args)
    {
        sopln(args[0]);
    }
}
```

NoteBook.

Op: Note Book.
space.

```
Java Test Note Book
```

↳ then we get only : Note.

6:34PM

6/2/22 Lecture 16 (16) Java Coding standards:-

Whenever we are writing Java code it is highly recommended to follow coding standard, whenever we are writing any component its name should reflect the purpose of that component (functionality). The main advantage of this approach is readability & maintainability of code will be improved

Ex 1)
```
class A
{
    public int m1(int x, int y)
    {
        return x+y;
    }
}
```

Camel case standard.

Ex (2)
```
package com.durgasoft.scjp;
public class Calculator
{
    public static int add(int number1,
                          int number2)
    {
        return number1+number2;
    }
}
```

Hi-technicity standard.

CLASSTIME Pg. No.
Date / /

* Coding standard for classes :-
Usually class names are nouns should starts with uppercase character and if it contains multiple words every inner words should start with uppercase character.

String
StringBuffer
Account
Dog
:
:

⟹ Nouns.

* Coding standard for Interface :-
Usually interface names are adjectives, should starts with uppercase character and if it contains multiple words every inner word should starts with uppercase characters.

for eg :- Runnable
Serializable
Comparable
:
:

⟹ Adjective.

* Coding standard for methods :-
Usually method names are either verbs or verb-noun combination.
should starts with lowercase alphabet symbol and if it contains multiple words then every innerword should starts with uppercase character (camelcase convention).

for eg:- { print() sleep() run() eat() } { start() } ⇒ Verbs

{ getName() setSalary() } ⇒ verb - Noun Combination

Verb Noun

Action

* Coding standard for Variable :-
Usually variable names are nouns. should starts with lowercase alphabet symbol and if it contains multiple words then every inner word should starts with uppercase character (camel case convention)

for eg:- { name age salary } { mobileNumber ; ; } ⇒ Noun.

* Coding standard for Constants :-
Usually constant names are nouns. should ~~starts with~~ Contain only uppercase characters and if it contains multiple words then these words are seperated with underscore (_) symbol.

for eg:- MAX_VALUE
MAX_PRIORITY

CLASSTIME Pg. No.
Date / /

NORM_PRIORITY
MIN_PRIORITY
PI
} ⇒ Nouns.

Note: Usually we can declare constants with public static and FINAL modifiers.

* JavaBean Coding Standards:
A JavaBean is a simple Java class with private properties and public getter and setter methods.

Ex:
```
public class StudentBean
{
    private String name;
    public void setName(String name)
    {
        this.name = name;
    }
    public String getName()
    {
        return name;
    }
}
```
↳ not void

class name ends with 'Bean' is not official convention from SUN Microsystems.

* Syntax for setter method:
(1) It should be public method.
(2) The return type should be void.

3.) method name should be prefixed with set.
4.) It should take some arguments, i.e. It should not be no arguments method.

* Syntax for getter method :-

(1) It should be public method.
(2) The Returntype should not be Void.
(3) Method name should prefixed with get.
(4) It should not take any argument.

Note :- For Boolean properties getter method name can be prefixed with either get or is. But recommended to use is.

eg:- private boolean empty;

public boolean getEmpty()
{
return empty;
}

public boolean isEmpty()
{
return empty;
}

CLASSTIME Pg. No.
Date / /

Coding standards for Listeners :-

To register a listener method name should be prefixed with Add and must be same

✓ public void (addMyActionListener(MyActionListener l).

✗ public void (registerMyActionListner(MyActionListener l) — invalid

✗ public void addMyActionListner(ActionListner l) — not same — invalid

Case 2 :- To unregister a Listner :-

✓ public void removeMyActionListner(MyActionListner l).

✗ public void (unRegisterMyActionListner(MyActionListner l).

✗ public void removeMyActionListner(ActionListner l).

✗ public void (deleteMyActionListner(MyActionListener l).

⇒ Method Name should be prefixed with (remove) only

End of Language Fundamentals (10).